

Java 面试宝典

目录

一 . Java 基础.....	1
1. Java 基础知识.....	1
2. Java 集合框架.....	22
3. Java 多线程.....	36
二. Java Web.....	54
1.JDBC 技术.....	54
2.网路通讯部分.....	55
3. Cookie 和 Session.....	56
4.Jsp 和 Servlet.....	57
5. Ajax &Jquery.....	58
三. 数据库.....	61
3.1 SQL 之连接查询.....	61
3.2 SQL 之聚合函数.....	62
3.3 SQL 之 SQL 注入.....	62
3.4 SQL Select 语句完整的执行顺序:.....	63
3.5 存储引擎.....	63
3.6 索引.....	65
3.7 数据库三范式.....	70

3.8 数据库事务.....	72
3.9 存储过程.....	75
3.10 触发器.....	75
3.11 数据库并发策略.....	76
3.12 数据库锁.....	79
3.13 基于 Redis 分布式锁.....	80
3.14 分区分表.....	81
3.15 应该使用哪一种方式来实施数据库分库分表 ,这要看数据库中数据量的 瓶颈 所在 , 并综合项目的业务类型进行考虑。	81
3.16 MySQL 读写分离.....	82
3.17 MySQL 常用 30 种 SQL 查询语句优化方法.....	83
3.18 数据库优化方案整理.....	88
四. SpringMVC 框架.....	102
4.1 什么是 SpringMVC ? 简单介绍下你对 SpringMVC 的理解?.....	102
4.2 SpringMVC 的流程 ?	102
4.3 SpringMVC 的优点.....	103
4.4 SpringMVC 的主要组件 ?	103
4.5 SpringMVC 和 Struts2 的区别有哪些?.....	104
4.6 SpringMVC 怎么样设定重定向和转发的 ?	104

4.7 SpringMVC 怎么和 Ajax 相互调用的 ?	105
4.8 如何解决 Post 请求中文乱码问题 , Get 的又如何处理呢 ?	105
4.9 SpringMVC 的异常处理 ?	106
4.10 SpringMVC 的控制器是不是单例模式,如果是,有什么问题,怎么解决 ?	106
4.11 SpringMVC 常用的注解有哪些 ?	106
4.12 SpringMVC 中的控制器的注解一般用那个,有没有别的注解可以替代 ?	107
4.13 如果在拦截请求中 , 我想拦截 get 方式提交的方法,怎么配置 ?	107
4.14 怎样在方法里面得到 Request,或者 Session ?	107
4.15 如果想在拦截的方法里面得到从前台传入的参数,怎么得到 ?	107
4.16 如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样 快速得到这个对象 ?	107
4.17 SpringMVC 中函数的返回值是什么 ?	107
4.18 SpringMVC 用什么对象从后台向前台传递数据的 ?	108
4.19 怎么样把 ModelMap 里面的数据放入 Session 里面 ?	108
4.20 SpringMVC 里面拦截器是怎么写的 ?	108
4.21 注解原理.....	109
五. Spring 框架.....	109
5.1 Spring 是什么?.....	109

5.2 Spring 的优点 ?	110
5.3 Spring 的 AOP 理解 ?	110
5.4 Spring 的 IOC 理解 ?	112
5.5 BeanFactory 和 ApplicationContext 有什么区别 ?	113
5.6 请解释 Spring Bean 的生命周期 ?	114
5.7 解释 Spring 支持的几种 bean 的作用域。	116
5.8 使用注解之前要开启自动扫描功能.....	116
5.9 Spring 框架中的单例 Beans 是线程安全的么 ?	118
5.10 Spring 如何处理线程并发问题 ?	118
5.11 Spring 基于 xml 注入 bean 的几种方式.....	118
5.12 Spring 的自动装配 :	119
5.13 Spring 框架中都用到了哪些设计模式 ?	120
5.14 Spring 事务的实现方式和实现原理.....	121
5.15 Spring 框架中有哪些不同类型的事件 ?	123
5.16 解释一下 Spring AOP 里面的几个名词.....	124
5.17 Spring 通知有哪些类型 ?	125
六. Mybatis 框架.....	127
6.1 什么是 Mybatis ?	127
6.2 Mybaits 的优点.....	128

6.3 MyBatis 框架的缺点.....	128
6.4 MyBatis 框架适用场合.....	128
6.5 MyBatis 与 Hibernate 有哪些不同？.....	128
6.6 #{}和\${}的区别是什么？.....	129
6.7 Mybatis 是如何进行分页的？分页插件的原理是什么？.....	129
6.8 Mybatis 是如何将 SQL 执行结果封装为目标对象并返回的？都有哪些映射形式？.....	130
6.9 Mybatis 动态 SQL 有什么用？执行原理？有哪些动态 SQL？.....	130
6.10 Xml 映射文件中,除了常见的 select insert update delete 标签之外,还有哪些标签？.....	130
6.11 Mybatis 的 Xml 映射文件中,不同的 Xml 映射文件, id 是否可以重复？.....	131
6.12 为什么说 Mybatis 是半自动 ORM 映射工具？它与全自动的区别在哪里？.....	131
6.13 MyBatis 实现一对一有几种方式?具体怎么操作的？.....	131
6.14 MyBatis 实现一对多有几种方式,怎么操作的？.....	131
6.15 Mybatis 是否支持延迟加载？如果支持，它的实现原理是什么？.....	132
6.16 Mybatis 的一级、二级缓存.....	132
6.17 什么是 MyBatis 的接口绑定？有哪些实现方式？.....	133

6.18 使用 MyBatis 的 mapper 接口调用时有哪些要求？	133
--	-----

6.19 简述 Mybatis 的插件运行原理，以及如何编写一个插件。	134
---	-----

七. SpringBoot SpringCloud 框架.....	135
--	------------

7.1 传统开发一般是基于什么框架进行开发的？	135
-------------------------------	-----

7.2 什么是 SOA 的？	136
----------------------	-----

7.3 什么是微服务？	136
-------------------	-----

7.4 SpringBoot 是什么？为什么我们选择使用 SpringBoot 开发？	140
---	-----

7.5 SpringCloud 是什么？	161
----------------------------	-----

7.6 SpringBoot 常见面试问题总结.....	177
------------------------------	-----

7.7 SpringCloud 常见面试问题总结.....	182
-------------------------------	-----

八. Saas 项目.....	188
------------------------	------------

1.名词与概念.....	188
--------------	-----

2.系统业务与系统环境.....	189
------------------	-----

九. 乐优商城.....	226
---------------------	------------

1. 项目介绍.....	226
--------------	-----

2. 项目架构.....	231
--------------	-----

3. 项目功能.....	233
--------------	-----

4. 模块业务.....	236
--------------	-----

5 . 商品管理.....	237
---------------	-----

6. 商品搜索.....	269
--------------	-----

7. 页面静态化.....	278
---------------	-----

8. 用户注册.....	279
--------------	-----

9. 用户登录.....	280
--------------	-----

10. 购物车.....	288
--------------	-----

11. 订单系统.....	301
---------------	-----

12.支付系统.....	302
--------------	-----

十. 十次方项目.....	304
---------------	-----

1. 十次方背景.....	304
---------------	-----

2. 十次方项目技术架构.....	316
-------------------	-----

3. 十次方项目面试常问问题.....	316
---------------------	-----

4. 十次方技术点.....	325
----------------	-----

一 . Java 基础

1. Java 基础知识

1.1 重载和重写的区别

重载： 发生在同一个类中，方法名必须相同，参数类型不同.个数不同.顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。

重写： 发生在父子类中，方法名.参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，

访问修饰符范围大于等于父类；如果父类方法访问修饰符为 `private` 则子类就不能重写该方法。

1.2 String 和 StringBuffer,StringBuilder 的区别是什么？String 为什么是不可变的？

可变性

简单的来说 :String 类中使用 `final` 关键字字符数组保存字符串，`private final char value[]`，所以 String 对象是不可变的。而 StringBuffer 与 StringBuilder 都继承自 `AbstractStringBuilder` 类，在 `AbstractStringBuilder` 中也是使用字符数组保存字符串 `char[]value` 但是没有用 `final` 关键字修饰，所以这两种对象都是可变的。

StringBuilder 与 StringBuffer 的构造方法都是调用父类构造方法也就是 `AbstractStringBuilder` 实现的，大家可以自行查阅源码。

AbstractStringBuilder.java

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {  
    char[] value;  
    int count;  
    AbstractStringBuilder() {  
    }  
    AbstractStringBuilder(int capacity) {  
        value = new char[capacity];  
    }  
}
```

线程安全性

String 中的对象是不可变的，也就可以理解为常量，线程安全。

AbstractStringBuilder 是 StringBuilder 与 StringBuffer 的公共父类，定义了一些字符串的基本操作，如 expandCapacity.append.insert.indexOf 等公共方法。StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder 并没有对

方法进行加同步锁，所以是非线程安全的。

性能

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {  
    char[] value;  
    int count;  
    AbstractStringBuilder() {  
    }  
    AbstractStringBuilder(int capacity) {  
        value = new char[capacity];  
    }  
}
```

每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象。

StringBuffer 每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 StirngBuilder 相比使用 StringBuffer 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

对于三者使用的总结：

1. 操作少量的数据 = String
2. 单线程操作字符串缓冲区下操作大量数据 = StringBuilder
3. 多线程操作字符串缓冲区下操作大量数据 = StringBuffer

1.3 自动装箱与拆箱

装箱：将基本类型用它们对应的引用类型包装起来；

拆箱：将包装类型转换为基本数据类型；

1.4 == 与 equals

==：它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象。（基本数据类型==比较的是值，引用数据类型==比较的是内存地址）

equals()：它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

情况 1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过 “==” 比较这两个对象。

情况 2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来两个对象的内容相等；若它们的内容相等，则返回 true（即，认为这两个对象相等）。

```
public class test1 {  
    public static void main(String[] args) {  
        String a = new String("ab"); // a 为一个引用  
        String b = new String("ab"); // b 为另一个引用,对象的内容一样  
        String aa = "ab"; // 放在常量池中  
        String bb = "ab"; // 从常量池中查找  
        if (aa == bb) // true  
            System.out.println("aa==bb");  
    }  
}
```

```
if (a == b) // false , 非同一对象
System.out.println("a==b");
if (a.equals(b)) // true
System.out.println("aEQb");
if (42 == 42.0) { // true
System.out.println("true");
}
}
}
```

说明：

String 中的 equals 方法是被重写过的，因为 object 的 equals 方法是比较的对象的内存地址，而 String 的 equals 方法比较的是对象的值。

当创建 String 类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 String 对象。

1.5 关于 final 关键字的一些总结

final 关键字主要用在三个地方：变量、方法、类。

1. 对于一个 final 变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。

2. 当用 final 修饰一个类时，表明这个类不能被继承。final 类中的所有成员方法都会被隐式地指定为 final 方法。

3. 使用 final 方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。

在早期的 Java 实现版本中，会将 final 方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升（现在的 Java 版本已经不需

要使用 final 方法进行这些优化了)。类中所有的 private 方法都隐式地指定为 final。

1.6 Object 类的常见方法总结

Object 类是一个特殊的类,是所有类的父类。它主要提供了以下 11 个方法:

方法 1: `public final native Class<?> getClass()`

// native 方法,用于返回当前运行时对象的 Class 对象,使用了 final 关键字修饰,故不允许子类重写。

方法 2: `public native int hashCode()`

// native 方法,用于返回对象的哈希码,主要使用在哈希表中,比如 JDK 中的 HashMap。

方法 3: `public boolean equals(Object obj)`

// 用于比较 2 个对象的内存地址是否相等, String 类对该方法进行了重写
用户比较字符串的值是否相等。

方法 4: `protected native Object clone() throws CloneNotSupportedException`

// native 方法,用于创建并返回当前对象的一份拷贝。一般情况下,对于任何对象 x,表达式 `x.clone() != x` 为 true, `x.clone().getClass() == x.getClass()` 为 true。Object 本身没有实现 Cloneable 接口,所以不重写 clone 方法并且进行调用的话会发生 CloneNotSupportedException 异常。

方法 5: `public String toString()`

// 返回类的名字@实例的哈希码的 16 进制的字符串。建议 Object 所有的子类都重写这个方法。

方法 6 : `public final native void notify()`

// native 方法，并且不能重写。唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念)。如果有多个线程在等待只会任意唤醒一个。

方法 7 : `public final native void notifyAll()`

// native 方法，并且不能重写。跟 notify 一样，唯一的区别就是会唤醒在此对象监视器上等待的所有线程，而不是一个线程。

方法 8 : `public final native void wait(long timeout) throws InterruptedException`

// native 方法，并且不能重写。暂停线程的执行。注意：sleep 方法没有释放锁，而 wait 方法释放了锁。timeout 是等待时间。

方法 9 : `public final void wait(long time, int nanos) throws InterruptedException`

// 多了 nanos 参数，这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。所以超时的时间还需要加上 nanos 毫秒。

方法 10 : `public final void wait() throws InterruptedException`

// 跟之前的 2 个 wait 方法一样，只不过该方法一直等待，没有超时时间这个概念。

方法 11 : `protected void finalize() throws Throwable { }`

// 实例被垃圾回收器回收的时候触发的操作。

1.7 Java 中的异常处理

在 Java 中，所有的异常都有一个共同的祖先 java.lang 包中的

Throwable 类。Throwable：有两个重要的子类：

Java异常类层次结构图



Exception（异常） 和 **Error（错误）**，二者都是 Java 异常处理的重要子类，各自都包含大量子类。

Error（错误）：是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java 虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java 虚拟机（JVM）一般会选择线程终止。

这些错误表示故障发生于虚拟机自身.或者发生在虚拟机试图执行应用时，如 Java 虚拟机运行错误（Virtual MachineError）.类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错

误，本质上也不应该试图去处理它所引起的异常状况。在 Java 中，错误通过 Error 的子类描述。

Exception (异常) :是程序本身可以处理的异常。Exception 类有一个重要的子类 **RuntimeException**。

RuntimeException 异常由 Java 虚拟机抛出。**NullPointerException**(要访问的变量没有引用任何对象时，抛出该 异常)。**ArithmeticException** (算术运算异常，一个整数除以 0 时，抛出该异常) 和

ArrayIndexOutOfBoundsException (下标越界异常)。

注意：异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。

Throwable 类常用方法

public String getMessage():返回异常发生时的详细信息

public String toString():返回异常发生时的简要描述

public String getLocalizedMessage():返回异常对象的本地化信息。使用 Throwable 的子类覆盖这个方法，可以声称本地化信息。如果子类没有覆盖该方法，则该方法返回的信息与 getMessage () 返回的结果相同

public void printStackTrace():在控制台上打印 Throwable 对象封装的异常信息

异常处理总结

try 块 :用于捕获异常。其后可接零个或多个 catch 块，如果没有 catch 块，则必须跟一个 finally 块。**catch 块 :**用于处理 try 捕获到的异常。

finally 块 :无论是否捕获或处理异常，finally 块里的语句都会被执行。

当在 try 块或 catch 块中遇到 return 语句

时，finally 语句块将在方法返回之前被执行。

在以下 4 种特殊情况下，finally 块不会被执行：

1. 在 finally 语句块中发生了异常。
2. 在前面的代码中用了 System.exit()退出程序。
3. 程序所在的线程死亡。
4. 关闭 CPU。

1.8 获取用键盘输入常用的的两种方法

方法 1：通过 Scanner

```
Scanner input = new Scanner(System.in);
```

```
String s = input.nextLine();
```

```
input.close();
```

方法 2：通过 BufferedReader

```
BufferedReader input = new BufferedReader(new  
InputStreamReader(System.in));
```

```
String s = input.readLine();
```

1.9 接口和抽象类的区别是什么

1. 接口的方法默认是 public，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现)，抽象类可以有非抽象的方法
2. 接口中的实例变量默认是 final 类型的，而抽象类中则不一定
3. 一个类可以实现多个接口，但最多只能实现一个抽象类

4. 一个类实现接口的话要实现接口的所有方法，而抽象类不一定

5. 接口不能用 new 实例化，但可以声明，但是必须引用一个实现该接口的对象 从设计层面来说，抽象是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

备注:在 JDK8 中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，必须重写，否则会报错。

1.10 JVM 调优及内存泄漏

1.10.1 JVM 调优工具

Jconsole , JProfile , VisualVM

Jconsole : JDK 自带，功能简单，但是可以在系统有一定负荷的情况下使用。对垃圾回收算法有很详细的跟踪。

JProfiler : 商业软件，需要付费。功能强大。

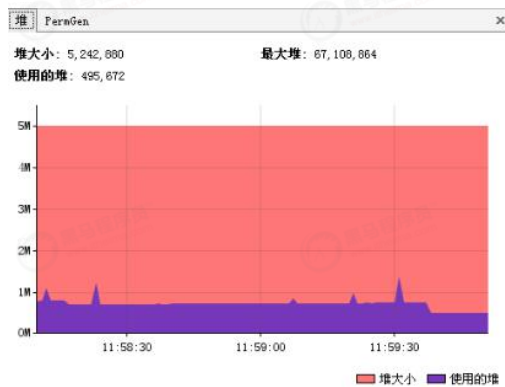
VisualVM : JDK 自带，功能强大，与 JProfiler 类似。推荐。

1.10.2 如何调优

观察内存释放情况.集合类检查.对象树

上面这些调优工具都提供了强大的功能 ,但是总的来说一般分为以下几类功能

堆信息查看



可查看堆空间大小分配（年轻代.年老代.持久代分配）

提供即时的垃圾回收功能

垃圾监控（长时间监控回收情况）

类名	实例数	实例大小	大小
java.lang.String	2416 (13%)	2416	57984 (64)
char[]	2161 (13%)	2161	236028 (234)
int[]	917 (5%)	917	294744 (234)
java.util.HashMap\$Entry	832 (5%)	832	24128 (14)
short[]	700 (4%)	700	31114 (14)
java.lang.Object[]	642 (4%)	642	21452 (14)
byte[]	564 (4%)	564	113652 (114)
java.util.HashMap\$Entry	560 (3%)	560	12632 (14)
java.lang.reflect.Method	412 (3%)	412	31724 (14)
java.lang.String[]	364 (3%)	364	6564 (14)
java.lang.Class[]	253 (2%)	253	3940 (14)
java.lang.Long	304 (2%)	304	4864 (14)
java.lang.ref.SoftReference	292 (2%)	292	9344 (14)
java.lang.Integer	264 (2%)	264	3408 (14)
java.util.HashMap\$Entry[]	273 (2%)	273	19552 (14)

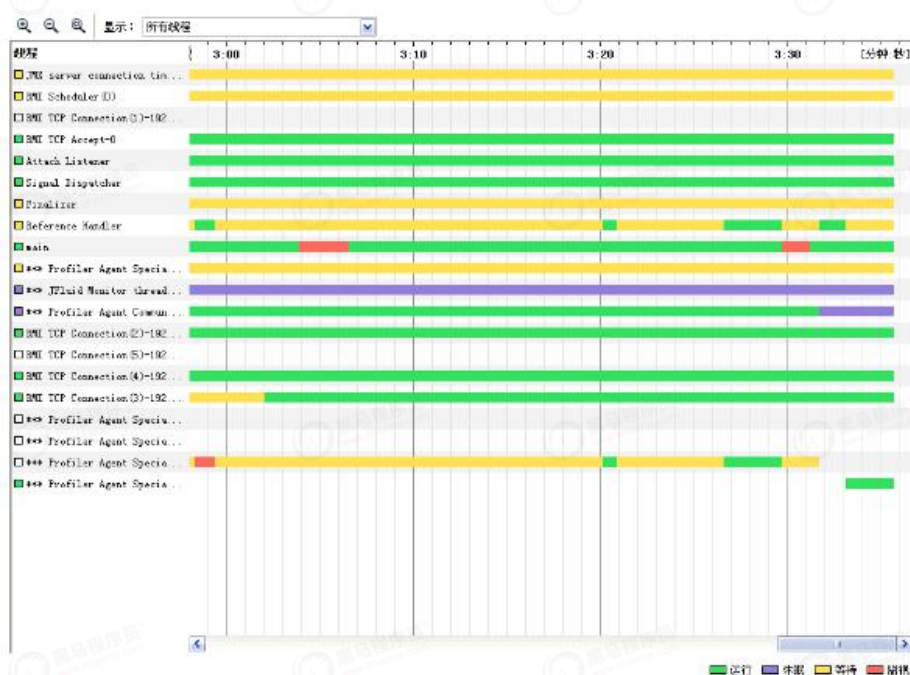
查看堆内类.对象信息查看：数量.类型等

对象引用情况查看

有了堆信息查看方面的功能，我们一般可以顺利解决以下问题：

- 年老代年轻代大小划分是否合理
- 内存泄漏
- 垃圾回收法设置是否合理

1.10.3 线程监控



线程信息监控：系统线程数量。

线程状态监控：各个线程都处在什么样的状态下

```

"Finalizer" daemon prio=8 tid=0x02ad0400 nid=0xd18 in Object.wait() [0x02c9f000..0x02c9fc94]
  java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x22e22b78> (a java.lang.ref.ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove(Unknown Source)
    - locked <0x22e22b78> (a java.lang.ref.ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove(Unknown Source)
    at java.lang.ref.Finalizer$FinalizerThread.run(Unknown Source)

  Locked ownable synchronizers:
    - None

"Reference Handler" daemon prio=10 tid=0x02ac5000 nid=0xb06 in Object.wait() [0x02c4f000..0x02c4fd14]
  java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x22e22c00> (a java.lang.ref.Reference$Lock)
    at java.lang.Object.wait(Object.java:485)
    at java.lang.ref.Reference$ReferenceHandler.run(Unknown Source)
    - locked <0x22e22c00> (a java.lang.ref.Reference$Lock)

  Locked ownable synchronizers:
    - None

"main" prio=8 tid=0x00068000 nid=0xc00 runnable [0x0093f000..0x0093f54]
  java.lang.Thread.State: RUNNABLE
    at CPUUserTest.longUseCpu(CPUUserTest.java:9)
    at CPUUserTest.main(CPUUserTest.java:31)

  Locked ownable synchronizers:
    - None

```

Dump 线程详细信息：查看线程内部运行情况

死锁检查

热点分析

Profiler

性能分析：

☒ CPU

☐ 内存

☐ 停止

状态：

应用程序已停止

性能分析结果



热点 - 方法	自用时间 [s]	自用时间	调用
java.lang.Thread.join (long)	4082 ns (38.9%)	2	
java.lang.ThreadGroup.add (Thread)	19.1 ns (0.9%)	2	
java.lang.Thread.start ()	3.62 ns (0.1%)	2	
java.util.logging.Logger.resetLogger (String)	1.95 ns (0%)	19	
java.util.logging.Logger.reset ()	1.54 ns (0%)	1	
java.util.logging.Logger.setLevel (java.util.logging.Level)	1.41 ns (0%)	19	
java.util.IdentityHashMap.keySet ()	1.29 ns (0%)	1	

CPU 热点：检查系统哪些方法占用的大量 CPU 时间

内存热点：检查哪些对象在系统中数量最大（一定时间内存活对象和销毁对象一起统计）

这两个东西对于系统优化很有帮助。我们可以根据找到的热点，有针对性的进行系统的瓶颈查找和进行系统优化，而不是漫无目的的进行所有代码的优化。

快照

快照是系统运行到某一时刻的一个定格。在我们进行调优的时候，不可能用眼睛去跟踪所有系统变化，依赖快照功能，我们就可以进行系统两个不同运行时刻，对象（或类、线程等）的不同，以便快速找到问题

举例说，我要检查系统进行垃圾回收以后，是否还有该收回的对象被遗漏下来的了。那么，我可以在进行垃圾回收前后，分别进行一次堆情况的快照，然后对比两次快照的对象情况。

1.10.4 内存泄漏检查

内存泄漏是比较常见的问题，而且解决方法也比较通用，这里可以重点说一下，而线程、热点方面的问题则是具体问题具体分析了。

内存泄漏一般可以理解为系统资源（各方面的资源，堆、栈、线程等）在错误使用的情况下，导致使用完毕的资源无法回收（或没有回收），从而导致新的资源分配请求无法完成，引起系统错误。

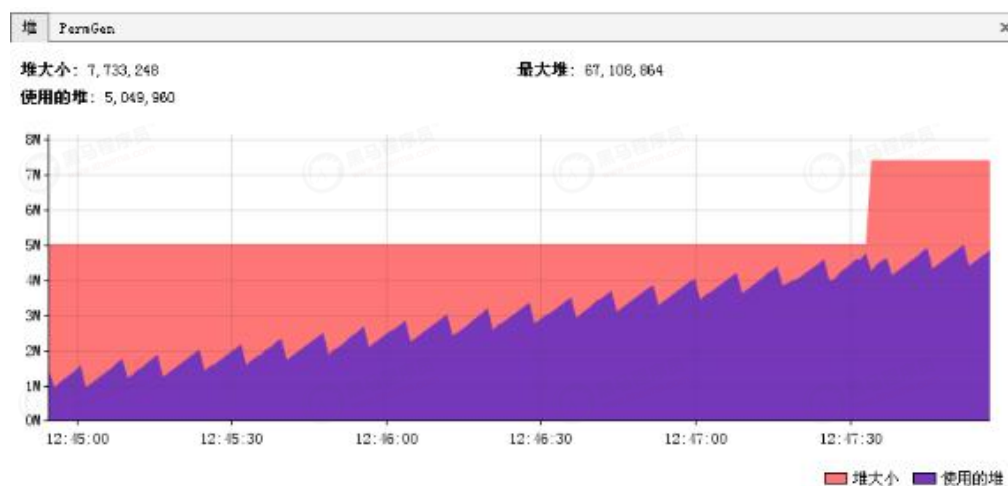
内存泄漏对系统危害比较大，因为他可以直接导致系统的崩溃。

需要区别一下，内存泄漏和系统超负荷两者是有区别的，虽然可能导致的最终结果是一样的。内存泄漏是用完的资源没有回收引起错误，而系统超负荷则是系统确实没有那么多资源可以分配了（其他的资源都在使用）。

老年代堆空间被占满

异常： java.lang.OutOfMemoryError: Java heap space

说明：



这是最典型的内存泄漏方式，简单说就是所有堆空间都被无法回收的垃圾对象占满，虚拟机无法再在分配新空间。

如上图所示，这是非常典型的内存泄漏的垃圾回收情况图。所有峰值部分都是一次垃圾回收点，所有谷底部分表示是一次垃圾回收后剩余的内存。连接所有谷底的点，可以发现一条由底到高的线，这说明，随时间的推移，系统的堆空间被不断占满，最终会占满整个堆空间。因此可以初步认为系统内部可能有内存泄漏。（上面的图仅供示例，在实际情况下收集数据的时间需要更长，比如几个小时或者几天）

解决 这种方式解决起来也比较容易，一般就是根据垃圾回收前后情况对比，同时根据对象引用情况（常见的集合对象引用）分析，基本都可以找到泄漏点。

持久代被占满

异常：java.lang.OutOfMemoryError: PermGen space

说明：Perm 空间被占满。无法为新的 class 分配存储空间而引发的异常。

这个异常以前是没有的，但是在 Java 反射大量使用的今天这个异常比较常见了。

主要原因就是大量动态反射生成的类不断被加载，最终导致 Perm 区被占满。

更可怕的是，不同的 classLoader 即便使用了相同的类，但是都会对其进行加载，相当于同一个东西，如果有 N 个 classLoader 那么他将会被加载 N 次。因此，某些情况下，这个问题基本视为无解。当然，存在大量 classLoader 和大量反射类的情况其实也不多。

解决：

1. -XX:MaxPermSize=16m
2. 换用 JDK。比如 JRocket

堆栈溢出

异常：java.lang.StackOverflowError

说明：这个就不多说了，一般就是递归没返回，或者循环调用造成

线程堆栈满

异常：Fatal: Stack size too small

说明：java 中一个线程的空间大小是有限制的。JDK5.0 以后这个值是 1M。与这个线程相关的数据将会保存在其中。但是当线程空间满了以后，将会出现上面异常。

解决：增加线程栈大小。-Xss2m。但这个配置无法解决根本问题，还要看代码部分是否有造成泄漏的部分。

系统内存被占满

异常：java.lang.OutOfMemoryError: unable to create new native thread

说明：这个异常是由于操作系统没有足够的资源来产生这个线程造成的。系统创建线程时，除了要在 Java 堆中分配内存外，操作系统本身也需要分配资源

来创建线程。因此，当线程数量大到一定程度以后，堆中或许还有空间，但是操作系统分配不出资源来了，就出现这个异常了。

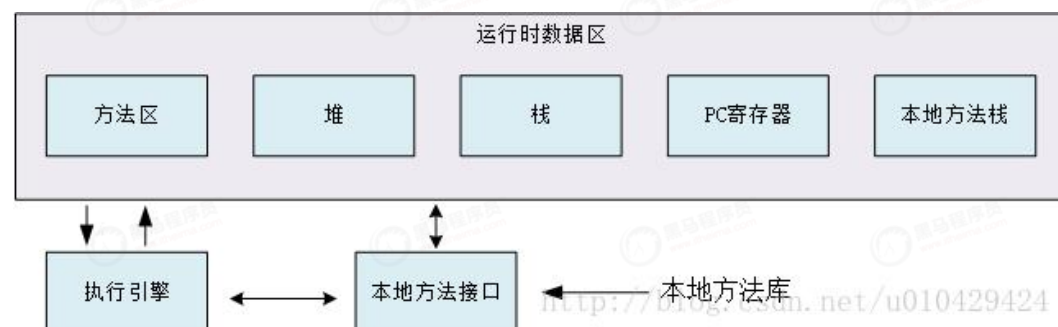
分配给 Java 虚拟机的内存愈多，系统剩余的资源就愈少，因此，当系统内存固定时，分配给 Java 虚拟机的内存越多，那么，系统总共能够产生的线程也就越少，两者成反比的关系。同时，可以通过修改-Xss 来减少分配给单个线程的空间，也可以增加系统总共内生产的线程数。

解决：

1. 重新设计系统减少线程数量。
2. 线程数量不能减少的情况下，通过-Xss 减小单个线程大小。以便能生产更多的线程。

1.11 JVM 内存管理

JVM 将内存划分为 6 个部分：PC 寄存器（也叫程序计数器）、虚拟机栈、堆、方法区、运行时常量池、本地方法栈。



PC 寄存器（程序计数器）：用于记录当前线程运行时的位置，每一个线程都有一个独立的程序计数器，线程的阻塞、恢复、挂起等一系列操作都需要程序计数器的参与，因此必须是线程私有的。

java 虚拟机栈：在创建线程时创建的，用来存储栈帧，因此也是线程私有

的。java 程序中的方法在执行时，会创建一个栈帧，用于存储方法运行时的临时数据和中间结果，包括局部变量表、操作数栈、动态链接、方法出口等信息。这些栈帧就存储在栈中。如果栈深度大于虚拟机允许的最大深度，则抛出 `StackOverflowError` 异常。

局部变量表：方法的局部变量列表，在编译时就写入了 class 文件

操作数栈：`int x = 1;` 就需要将 1 压入操作数栈，再将 1 赋值给变量 x

java 堆：java 堆被所有线程共享，堆的主要作用就是存储对象。如果堆空间不够，但扩展时又不能申请到足够的内存时，则抛出 `OutOfMemoryError` 异常。

StackOverflowError	OutOfMemoryError
java 栈	java 堆
栈深度超过范围了(比如 递归层数太多了)	内存空间不够了(需要及时释放内存)

方法区：方法区被各个线程共享，用于存储静态变量、运行时常量池等信息。

本地方法栈：本地方法栈的主要作用就是支持 native 方法，比如在 java 中调用 C/C++

1.12 GC 回收机制

1. 哪些内存需要回收？—— who
2. 什么时候回收？—— when
3. 怎么回收？—— how

1. 哪些内存需要回收？

Java 堆方法区的内存

线程私有	线程共享
程序计数器.虚拟机栈.本地方法栈	java 堆 方法区
随线程生而生，随线程去而去。线程分配多少内存都是有数的，当线程销毁时，内存就被释放了	堆和方法区的内存都是动态分配的（使用 new 关键字），所以也需要动态回收。 这部分内存的回收依赖 GC 完成

2. 什么时候回收？

引用计数法

可达性分析

2.1 引用计数法

给对象添加一个引用计数器，每当有一个地方引用它时，计数器加一。反之每当一个引用失效时，计数器减一。当计数器为 0 时，则表示对象不被引用。

举个例子：

```
Object a = new Object(); // a 的引用计数为 1  
  
a = null; // a 的引用计数为 0，等待 GC 回收
```

但是，引用计数法不能解决对象之间的循环引用，见下例

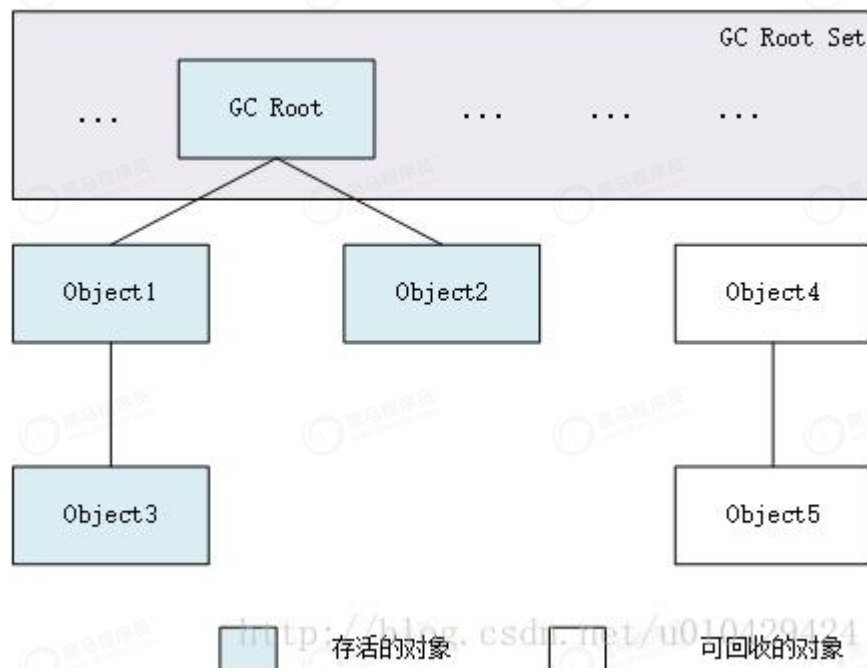
```
Object a = new Object(); // a 的引用计数为 1  
  
Object b = new Object(); // b 的引用计数为 1  
  
a.next = b; // a 的引用计数为 2  
  
b.next = a; // b 的引用计数为 2
```

`a = null;` // a 的引用计数为 1, 尽管已经显示地将 a 赋值为 null, 但是由于引用计数为 1, GC 无法回收 a

`b = null;` // b 的引用计数为 1, 同理, GC 也不回收 b

2.2 可达性分析

设立若干根对象 (GC Root), 每个对象都是一个子节点, 当一个对象找不到根时, 就认为该对象不可达。



没有一条从根到 Object4 和 Object5 的路径, 说明这两个对象到根是不可达的, 可以被回收

补充: java 中, 可以作为 GC Roots 的对象包括:

- java 虚拟机栈中引用的对象
- 方法区中静态变量引用的对象
- 方法区中常量引用的对象
- 本地方法栈中引用的对象

3. 怎么回收？

标记——清除算法

复制算法

分代算法

3.1 标记——清除算法

遍历所有的 GC Root，分别标记处可达的对象和不可达的对象，然后将不可达的对象回收。

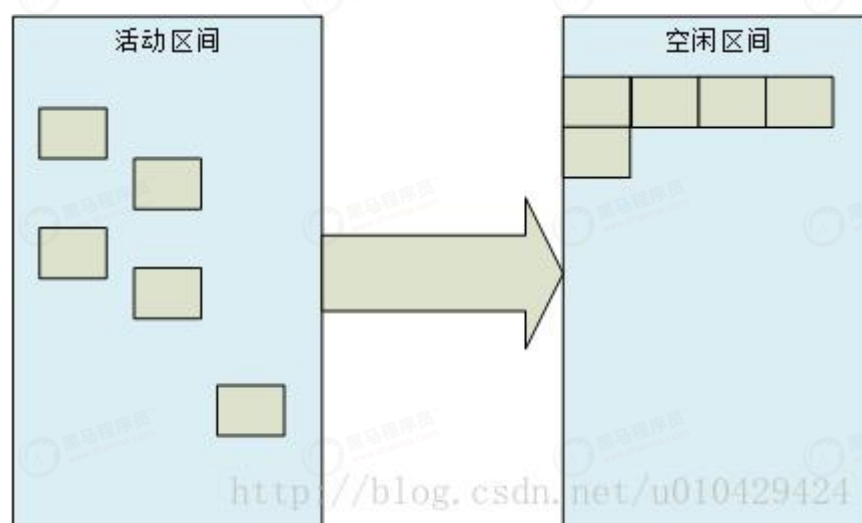
缺点是：效率低.回收得到的空间不连续

3.2 复制算法

将内存分为两块，每次只使用一块。当这一块内存满了，就将还存活的对象复制到另一块上，并且严格按照内存地址排列，然后把已使用的那块内存统一回收。

优点是：能够得到连续的内存空间

缺点是：浪费了一半内存



3.3 分代算法

在 java 中，把内存中的对象按生命长短分为：

新生代：活不了多久就 go die 了，比如局部变量

老年代：老不死的，活的久但也会 go die，比如一些生命周期长的对象

永久代：千年王八万年龟，不死，比如加载的 class 信息

有一点需要注意：新生代和老年代存储在 java 虚拟机堆上；永久代存储在方法区上

	回收方法
新生代	使用复制算法
老年代	使用标记——清除算法
永久代	_____

补充：java finalize()方法：

在被 GC 回收前，可以做一些操作，比如释放资源。有点像析构函数，但是一个对象只能调用一次 finalize()方法。

2. Java 集合框架

2.1 ArrayList 与 LinkedList 异同

1. 是否保证线程安全：ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；

2. 底层数据结构：ArrayList 底层使用的是 Object 数组；LinkedList 底层使用的是双向链表数据结构（JDK1.6 之前为循环链表，JDK1.7 取消了循环。注意双向链表和双向循环链表的区别：）；

3. 插入和删除是否受元素位置的影响： ① **ArrayList** 采用数组存储，所以插入和删除元素的时间复杂度受元素

位置的影响。 比如：执行 `add(E e)` 方法的时候，**ArrayList** 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是 $O(1)$ 。但是如果要在指定位置 `i` 插入和删除元素的话（`add(int index, E element)`）时间复杂度就为 $O(n-i)$ 。因为在进行上述操作的时候集合中第 `i` 和第 `i` 个元素之后的 $(n-i)$ 个元素都要执行向后位/向前移一位的操作。② **LinkedList** 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 $O(1)$ 而数组为近似 $O(n)$ 。

4. 是否支持快速随机访问： **LinkedList** 不支持高效的随机元素访问，而 **ArrayList** 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于 `get(int index)` 方法)。

5. 内存空间占用： **ArrayList** 的空间浪费主要体现在在 `list` 列表的结尾会预留一定的容量空间，而 **LinkedList** 的空间花费则体现在它的每一个元素都需要消耗比 **ArrayList** 更多的空间（因为要存放直接后继和直接前驱以及数据）。

补充内容:RandomAccess 接口

```
public interface RandomAccess {  
  
}
```

查看源码我们发现实际上 **RandomAccess** 接口中什么都没有定义。所以，在我看来 **RandomAccess** 接口不过是一个标识罢了。标识什么？标识实现这个接口的类具有随机访问功能。

在 `binarySearch ( )` 方法中，它要判断传入的 `list` 是否 `RandomAccess` 的实例，如果是，调用

`indexedBinarySearch ( )` 方法，如果不是，那么调用 `iteratorBinarySearch ( )` 方法

```
public static <T>
int binarySearch(List<? extends Comparable<? super T>> list, T key) {
    if (list instanceof RandomAccess || list.size() < BINARYSEARCH_THRESHOLD)
        return Collections.indexedBinarySearch(list, key);
    else
        return Collections.iteratorBinarySearch(list, key);
}
```

`ArrayList` 实现了 `RandomAccess` 接口，而 `LinkedList` 没有实现。为什么呢？我觉得还是和底层数据结构有关！`ArrayList` 底层是数组，而 `LinkedList` 底层是链表。数组天然支持随机访问，时间复杂度为 $O(1)$ ，所以称为快速随机访问。链表需要遍历到特定位置才能访问特定位置的元素，时间复杂度为 $O(n)$ ，所以不支持快速随机访问。

`ArrayList` 实现了 `RandomAccess` 接口，就表明了他具有快速随机访问功能。`RandomAccess` 接口只是标识，并不是说 `ArrayList` 实现 `RandomAccess` 接口才具有快速随机访问功能的！

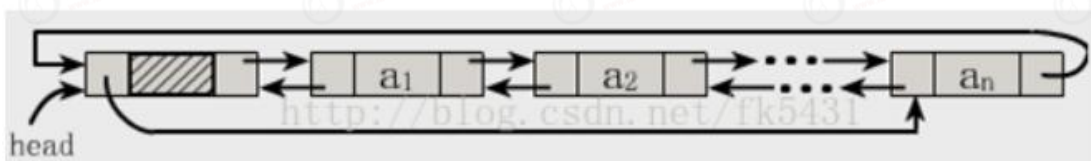
下面再总结一下 `list` 的遍历方式选择：

实现了 `RandomAccess` 接口的 `list`，优先选择普通 `for` 循环，其次 `foreach`，未实现 `RandomAccess` 接口的 `list`，优先选择 `iterator` 遍历（`foreach` 遍历底层也是通过 `iterator` 实现的），大 `size` 的数据，千万不要使用普通 `for` 循环。补充：数据结构基础之双向链表

双向链表也叫双链表，是链表的一种，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。一般我们都构造双向循环链表，如

下图所示 ,同时下图也是 LinkedList 底层使用的是双向循环链表数据结构。

```
public interface RandomAccess {  
    }  
    public static <T>  
    int binarySearch(List<? extends Comparable<? super T>> list, T key) {  
        if (list instanceof RandomAccess || list.size() < BINARYSEARCH_THRESHOLD)  
            return Collections.indexedBinarySearch(list, key);  
        else  
            return Collections.iteratorBinarySearch(list, key);  
    }  
}
```



2.2 ArrayList 与 Vector 区别

Vector 类的所有方法都是同步的。可以由两个线程安全地访问一个 Vector 对象,但是一个线程访问 Vector 的话代码要在同步操作上耗费大量的时间。

ArrayList 不是同步的 ,所以在不需要保证线程安全时时建议使用 ArrayList。

2.3 HashMap 的底层实现

JDK1.8 之前

JDK1.8 之前 HashMap 底层是 **数组和链表** 结合在一起使用也就是 **链表散列**。HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值 , 然后通过 $(n - 1) \& \text{hash}$ 判断当前元素存放的位置 (这里的 n 指的是数组的长度) , 如果当前位置存在元素的话 , 就判断该元素与要存入的元素的 hash 值以及 key 是否相同 , 如果相同的话 , 直接覆盖 , 不相同就通过拉链表解决冲突。

所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。

JDK 1.8 HashMap 的 hash 方法源码:

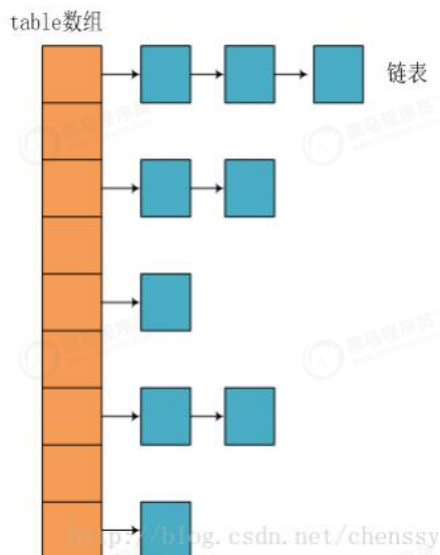
JDK 1.8 的 hash 方法 相比于 JDK 1.7 hash 方法更加简化，但是原理不变。

```
static final int hash(Object key) {  
  
    int h;  
  
    // key.hashCode() : 返回散列值也就是 hashCode  
  
    // ^ : 按位异或  
  
    // >>>:无符号右移，忽略符号位，空位都以 0 补齐  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}
```

对比一下 JDK1.7 的 HashMap 的 hash 方法源码.

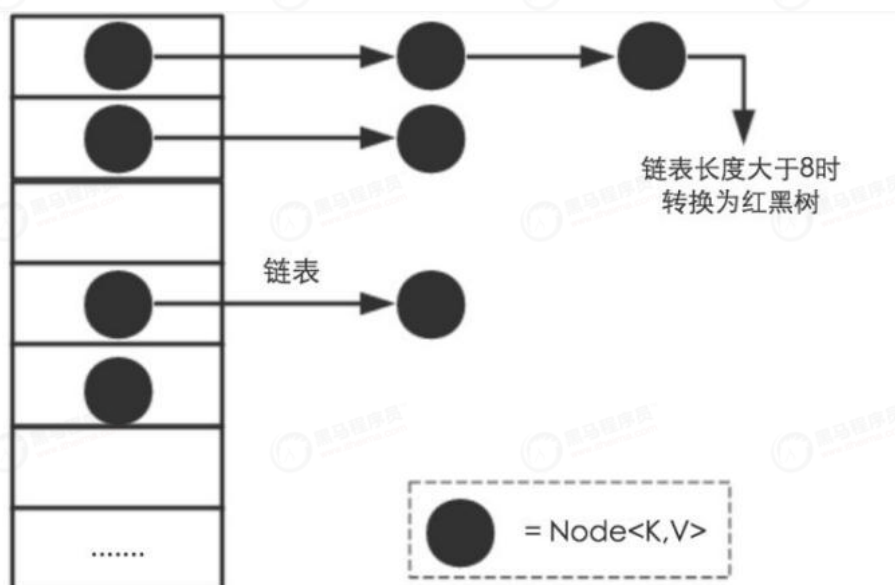
```
static int hash(int h) {  
  
    // This function ensures that hashCodes that differ only by  
    // constant multiples at each bit position have a bounded  
    // number of collisions (approximately 8 at default load factor).  
  
    h ^= (h >>> 20) ^ (h >>> 12);  
    return h ^ (h >>> 7) ^ (h >>> 4);  
}
```

相比于 JDK1.8 的 hash 方法，JDK 1.7 的 hash 方法的性能会稍差一点点，因为毕竟扰动了 4 次。所谓“**拉链法**”就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。



JDK1.8 之后

相比于之前的版本，JDK1.8 之后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间。



TreeMap、TreeSet以及JDK1.8之后的HashMap底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

2.4 HashMap 和 Hashtable 的区别

1. **线程是否安全**：HashMap 是非线程安全的，Hashtable 是线程安全的；Hashtable 内部的方法基本都经过

synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；

2. **效率**：因为线程安全的问题，HashMap 要比 Hashtable 效率高一点。另外，Hashtable 基本被淘汰，不要在代码中使用它；

3. **对 Null key 和 Null value 的支持**：HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键

所对应的值为 null。。但是在 Hashtable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。

4. **初始容量大小和每次扩充容量大小的不同**：①创建时如果不指定容量初始值，Hashtable 默认的初始大小为 11，之后每次扩充，容量变为原来的 $2n+1$ 。HashMap 默认的初始化大小为 16。之后每次扩充，容量变为原来的 2 倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为 2 的幂次方大小（HashMap 中的 tableSizeFor() 方法保证，下面给出了源代码）。也就是说 HashMap 总是使用 2 的幂作为哈希表的大小，后面会介绍到为什么是 2 的幂次方。

5. **底层数据结构**：JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

HashMap 中带有初始容量的构造函数：

```

public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
            initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
            loadFactor);
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}

public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}

```

下面这个方法保证了 HashMap 总是使用 2 的幂作为哈希表的大小。

```

/**
 * Returns a power of two size for the given target capacity.
 */
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}

```

2.5 HashMap 的长度为什么是 2 的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值-2147483648 到 2147483647，前后加起来大概 40 亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个 40 亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法

是 “ $(n - 1) \& \text{hash}$ ”。（ n 代表数组长度）。这也就解释了 HashMap 的长度为什么是 2 的幂次方。

这个算法应该如何设计呢？

我们首先可能会想到采用 % 取余的操作来实现。但是，重点来了：“取余(%) 操作中如果除数是 2 的幂次则等价于与其

除数减一的与(&)操作（也就是说 $\text{hash} \% \text{length} = \text{hash} \& (\text{length} - 1)$ 的前提是 length 是 2 的 n 次方；）。” 并且采

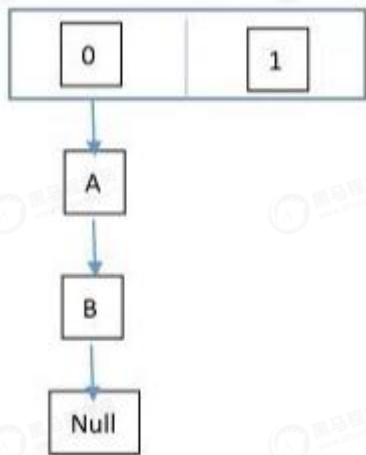
用二进制位操作 &，相对于 % 能够提高运算效率，这就解释了 HashMap 的长度为什么是 2 的幂次方。

2.6 HashMap 多线程操作导致死循环问题

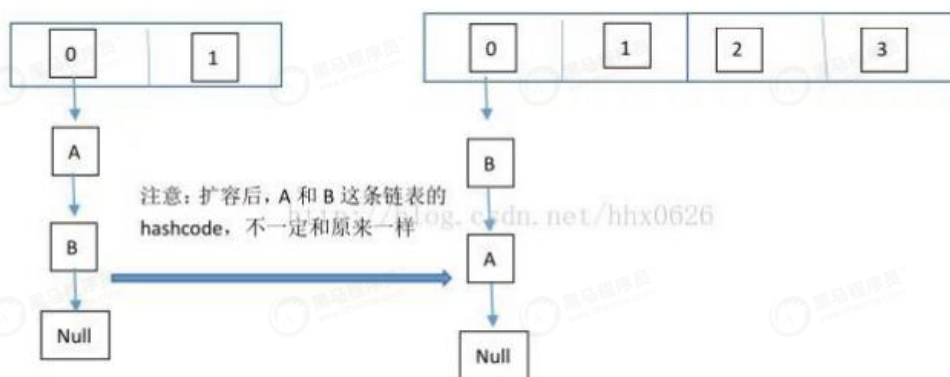
在多线程下，进行 put 操作会导致 HashMap 死循环，原因在于 HashMap 的扩容 `resize()` 方法。由于扩容是新建一个数组，复制原数据到数组。由于数组下标挂有链表，所以需要复制链表，但是多线程操作有可能导致环形链表。复制链表过程如下：

以下模拟 2 个线程同时扩容。假设，当前 HashMap 的空间为 2（临界值为 1），hashcode 分别为 0 和 1，在散列地址 0 处有元素 A 和 B，这时候要添加元素 C，C 经过 hash 运算，得到散列地址为 1，这时候由于超过了临界值，空间不够，需要调用 `resize` 方法进行扩容，那么在多线程条件下，会出现条件竞争，模拟过程如下：

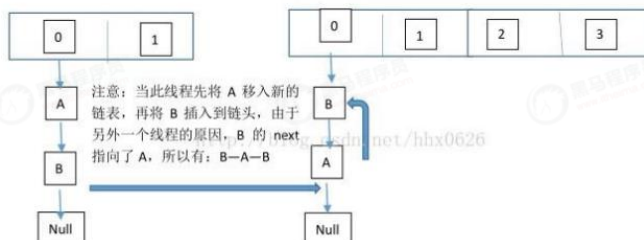
线程一：读取到当前的 HashMap 情况，在准备扩容时，线程二介入



线程二：读取 HashMap，进行扩容



线程一：继续执行



这个过程为，先将 A 复制到新的 hash 表中，然后接着复制 B 到链头（A 的前边：B.next=A），本来 B.next=null，到此也就结束了（跟线程二一样的过程），但是，由于线程二扩容的原因，将 B.next=A，所以，这里继续复制 A，让 A.next=B，由此，环形链表出现：B.next=A; A.next=B

注意：JDK1.8 已经解决了死循环的问题。

2.7 HashSet 和 HashMap 区别

如果你看过 HashSet 源码的话就应该知道：HashSet 底层就是基于 HashMap 实现的。（ HashSet 的源码非常非常少，因为除了 clone() 方法.writeObject()方法.readObject()方法是 HashSet 自己不得不实现之外，其他方法都是直接调用 HashMap 中的方法。）

HashMap	HashSet
实现了Map接口	实现Set接口
存储键值对	仅存储对象
调用put () 向map中添加元素	调用add () 方法向Set中添加元素
HashMap使用键 (Key) 计算Hashcode	HashSet使用成员对象来计算hashcode值， 对于两个对象来说hashcode可能相同， 所以equals()方法用来判断对象的相等性， 如果两个对象不同的话，那么返回false
HashMap相对于HashSet较快，因为它是使用唯一的键获取对象	HashSet较HashMap来说比较慢

2.8 ConcurrentHashMap 和 Hashtable 的区别

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

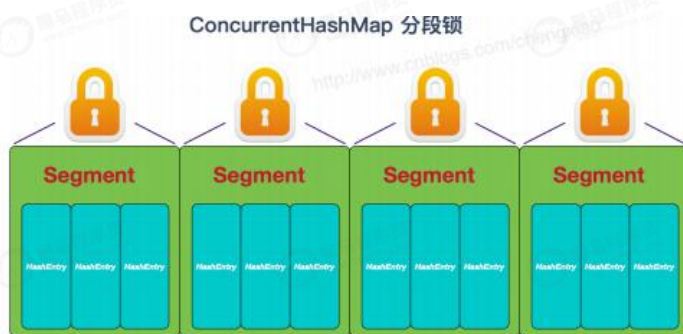
底层数据结构： JDK1.7 的 ConcurrentHashMap 底层采用 **分段的数组+链表** 实现，JDK1.8 采用的数据结构跟 HashMap1.8 的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 **数组+链表** 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；

实现线程安全的方式（重要）： ① 在 JDK1.7 的时候，**ConcurrentHashMap（分段锁）** 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。到了 JDK1.8 的时候已经摒弃了 Segment

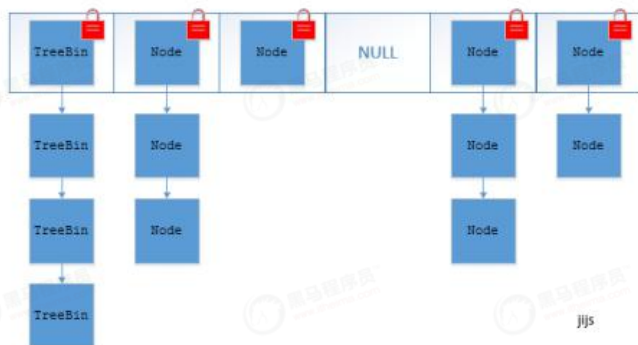
HashTable 全表锁

```

graph TD
    Lock[Padlock] --- Slot1
    Lock --- Slot4
    subgraph Row
        Slot1["Entry  
key  
value  
hash  
next"]
        Slot2["Entry  
key  
value  
hash  
next"]
        Slot3["NULL"]
        Slot4["Entry  
key  
value  
hash  
next"]
        Slot5["Entry  
key  
value  
hash  
next"]
        Slot6["NULL"]
    end
    Slot1 -- next --> Null1[null]
    Slot2 -- next --> Slot2_2["Entry  
key  
value  
hash  
next"]
    Slot3 -- next --> Null2[null]
    Slot4 -- next --> Slot4_2["Entry  
key  
value  
hash  
next"]
    Slot5 -- next --> Null3[null]
    Slot2_2 -- next --> Slot2_3["Entry  
key  
value  
hash  
next"]
    Slot4_2 -- next --> Null4[null]
    Slot2_3 -- next --> Null5[null]
  
```



JDK1.8的ConcurrentHashMap (TreeBin: 红黑二叉树节点 Node: 链表节点) :



2.9 ConcurrentHashMap 线程安全的具体实现方式/底层具体实现

JDK1.7 (上面有示意图)

首先将数据分为一段一段的存储,然后给每一段数据配一把锁,当一个线程占用锁访问其中一个段数据时,其他段的数据也能被其他线程访问。

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。

Segment 实现了 ReentrantLock,所以 Segment 是一种可重入锁,扮演锁的角色。HashEntry 用于存储键值对数据。

```
static class Segment<K,V> extends ReentrantLock implements
Serializable { }
```

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和 HashMap 类似,是一种数组和链表结构,一个 Segment 包含一个

HashEntry 数组,每个 HashEntry 是一个链表结构的元素,每个 Segment 守护着一个 HashEntry 数组里的元素,当对 HashEntry 数组的数据进行修改时,必须首先获得对应的 Segment 的锁。

JDK1.8 (上面有示意图)

ConcurrentHashMap 取消了 Segment 分段锁,采用 CAS 和 synchronized 来保证并发安全。数据结构跟 HashMap1.8 的结构类似,数组+链表/红黑二叉树。

synchronized 只锁定当前链表或红黑二叉树的首节点,这样只要 hash 不冲突,就不会产生并发,效率又提升 N 倍。

2.10 集合框架底层数据结构总结

Collection

1. List

ArrayList: Object 数组

Vector: Object 数组

LinkedList: 双向链表(JDK1.6 之前为循环链表, JDK1.7 取消了循环) 详细可阅读 [JDK1.7-LinkedList 循环链表优化](#)

2. Set

HashSet(无序,唯一): 基于 HashMap 实现的,底层采用 HashMap 来保存元素

LinkedHashSet: LinkedHashSet 继承与 HashSet,并且其内部是通过 LinkedHashMap 来实现的。有点类

似于我们之前说的 LinkedHashMap 其内部是基于 HashMap 实现一样，不过还是有一点点区别的。

TreeSet (有序，唯一)： 红黑树(自平衡的排序二叉树。)

2.Map

HashMap： JDK1.8 之前 HashMap 由数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）。JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间

LinkedHashMap： LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外，LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。详细可以查看：

HashTable： 数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的

TreeMap： 红黑树（自平衡的排序二叉树）

3. Java 多线程

关于 Java 多线程，在面试的时候，问的比较多的就是①**悲观锁和乐观锁**（具体可以看我的这篇文章：[面试必备之乐观锁与悲观锁](#)）。②**synchronized 和 lock 区别以及 volatile 和 synchronized 的区别**，③**可重入锁与非可重入锁的区别**。④**多线程是解决什么问题的**。⑤**线程池解决什么问题**。⑥**线程池的原理**。⑦

线程池使用时的注意事项.⑧AQS 原有没有在项目中实际使用多线程的经历。所以,如果你在的项目中有实际使用 Java 多线程的经历 的话,会为你加分不少哦!

3.1 synchronized 关键字的 5 连击

3.1.1 说一说自己对于 synchronized 关键字的了解

synchronized 关键字解决的是多个线程之间访问资源的同步性, synchronized 关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。另外,在 Java 早期版本中, synchronized 属于重量级锁,效率低下,因为监视器锁 (monitor) 是依赖于底层的操作系统的 Mutex Lock 来实现的, Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程,都需要操作系统帮忙完成,而操作系统实现线程之间的切换时需要从用户态转换到内核态,这个状态之间的转换需要相对比较长的时间,时间成本相对较高,这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对 synchronized 较大优化,所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6 对锁的实现引入了大量的优化,如自旋锁.适应性自旋锁.锁消除.锁粗化.偏向锁.轻量级锁等技术来减少锁操作的开销。

3.1.2 说说自己是怎么使用 synchronized 关键字, 在项目中用到 了吗?

synchronized 关键字最主要的三种使用方式 :

修饰实例方法，作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁

修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁。 也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（`static` 表明这是该类的一个静态资源，不管 `new` 了多少个对象，只有一份，所以对该类的所有对象都加了锁）。所以如果一个线程 A 调用一个实例对象的非静态 `synchronized` 方法，而线程 B 需要调用这个实例对象所属类的静态 `synchronized` 方法，是允许的，不会发生互斥现象，**因为访问静态 `synchronized` 方法占用的锁是当前类的锁，而访问非静态 `synchronized` 方法占用的锁是当前实例对象锁。**

修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。 和 `synchronized` 方法一样，`synchronized(this)` 代码块也是锁定当前对象的。`synchronized` 关键字加到 `static` 静态方法和 `synchronized(class)` 代码块上都是给 `Class` 类上锁。这里再提一下：`synchronized` 关键字加到非 `static` 静态方法上是给对象实例上锁。另外需要注意的是：尽量不要使用 `synchronized(String a)` 因为 JVM 中，字符串常量

池具有缓冲功能！下面我已一个常见的面试题为例讲解一下 `synchronized` 关键字的具体使用。

面试中面试官经常会说：“单例模式了解吗？来给我手写一下！给我解释一下双重检验锁方式实现单例模式的原理呗！”

双重校验锁实现对象单例（线程安全）

```
public class Singleton {  
    private volatile static Singleton uniqueInstance;
```

```
private Singleton() {  
}  
public static Singleton getUniqueInstance() {
```

//先判断对象是否已经实例过，没有实例化过才进入加锁代码另外，需要注意 uniqueInstance 采用 volatile 关键字修饰也是很有必要。

uniqueInstance 采用 volatile 关键字修饰也是很有必要的， uniqueInstance = new Singleton(); 这段代码其实是分

为三步执行：

1. 为 uniqueInstance 分配内存空间
2. 初始化 uniqueInstance
3. 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被

初始化。使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

3.1.3 讲一下 synchronized 关键字的底层原理

synchronized 关键字底层原理属于 JVM 层面。

① synchronized 同步语句块的情况

通过 JDK 自带的 javap 命令查看 SynchronizedDemo 类的相关字节码信息：首先切换到类的对应目录执行 javac SynchronizedDemo.java 命令生成编译后的 .class 文件，然后执行 javap -c -s -v -l SynchronizedDemo.class。

```
if (uniqueInstance == null) {  
    //类对象加锁  
    synchronized (Singleton.class) {
```

```
if (uniqueInstance == null) {
    uniqueInstance = new Singleton();
}
}
return uniqueInstance;
}
}

public class SynchronizedDemo {
    public void method() {
        synchronized (this) {
            System.out.println("synchronized 代码块");
        }
    }
}
```

从上面我们可以看出：

synchronized 同步语句块的实现使用的是 monitorenter 和

monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。 当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor 对象存在于每个 Java 对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么 Java 中任意对象可以作为锁的原因) 的持有权.当计数器为 0 则可以成功获取，获取后将锁计数器设为 1 也就是加 1。相应的在执行 monitorexit 指令后，将锁计数器设为 0，表明锁被释放。如果获取对象锁失败，那当

前线程就要阻塞等待，直到锁被另外一个线程释放为止。

② synchronized 修饰方法的的情况

```
public class SynchronizedDemo2 {
    public synchronized void method() {
        System.out.println("synchronized 方法");
    }
}
```

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是

ACC_SYNCHRONIZED 标识,该标识指明了该方法是一个同步方法,JVM 通过该 ACC_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法,从而执行相应的同步调用。

3.1.4 说说 JDK1.6 之后的 synchronized 关键字底层做了哪些优化,可以详细介绍一下这些优化吗?

JDK1.6 对锁的实现引入了大量的优化,如偏向锁.轻量级锁.自旋锁.适应性自旋锁.锁消除.锁粗化等技术来减少锁操作的开销。

锁主要存在四种状态,依次是:无锁状态.偏向锁状态.轻量级锁状态.重量级锁状态,他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级,这种策略是为了提高获得锁和释放锁的效率。

3.1.5 谈谈 synchronized 和 ReentrantLock 的区别

(1)两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是:自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁,此时这个对象锁还没有释放,当其再次想要获取这个对象的锁的时候还是可以获取的,如果不可锁重入的话,就会造成死锁。同一个线程每次获取锁,锁的计数器都自增 1,所以要等到锁的计数器下降为 0 时才能释放锁。

(2) synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API

synchronized 是依赖于 JVM 实现的,前面我们也讲到了虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化,但是这些优化都是在虚拟机层面实现的,并没有直接暴露给我们。ReentrantLock 是 JDK 层面实现的(也就是 API 层面,需要 lock() 和 unlock 方法配合 try/finally 语句块来完成),所以我们可以查看它的源代码,来看它是如何实现的。

(3)ReentrantLock 比 synchronized 增加了一些高级功能

相比 `synchronized` , `ReentrantLock` 增加了一些高级功能。主要来说主要有三点 : **等待可中断 ;**

可实现公平锁 ; 可实现选择性通知 (锁可以绑定多个条件) `ReentrantLock` 提供了一种能够中断等待锁的线程的机制 , 通过 `lock.lockInterruptibly()` 来实现这个机制。也就是说正在等待的线程可以选择放弃等待 , 改为处理其他事情。

`ReentrantLock` 可以指定是公平锁还是非公平锁。而 `synchronized` 只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。 `ReentrantLock` 默认情况是非公平的 , 可以通过 `ReentrantLock` 类的 `ReentrantLock(boolean fair)` 构造方法来制定是否是公平的。 `synchronized` 关键字与 `wait()` 和 `notify/notifyAll()` 方法相结合可以实现等待/通知机制 , `ReentrantLock` 类当然也可以实现 , 但是需要借助于 `Condition` 接口与 `newCondition()` 方法。 `Condition` 是 JDK1.5 之后才有的 , 它具有很

好的灵活性 , 比如可以实现多路通知功能也就是在一个 `Lock` 对象中可以创建多个 `Condition` 实例 (即对象监视器) , 线程对象可以注册在指定的 `Condition` 中 , 从而可以有选择性的进行线程通知 , 在调度线程上更加灵活。 在使用 `notify/notifyAll()` 方法进行通知时 , 被通知的线程是由 JVM 选择的 , 用 `ReentrantLock` 类结合 `Condition` 实例可以实现 “选择性通知” , 这个功能非常重要 , 而且是 `Condition` 接口默认提供的。而 `synchronized` 关键字就相当于整个 `Lock` 对象中只有一个 `Condition` 实例 , 所有的线程都注册在它一个身上。如果执行 `notifyAll()` 方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题 , 而 `Condition` 实例的 `signalAll()` 方法 只会唤醒注册在该 `Condition` 实例中的所有等待线程。如果你想使用上述功能 , 那么选择 `ReentrantLock` 是一个不错的选择。

3.2 线程池的 4 连击

3.2.1 讲一下 Java 内存模型

在 JDK1.2 之前，Java 的内存模型实现总是从**主存（即共享内存）**读取变量，是不需要进行特别的注意的。而在当前的 Java 内存模型下，线程可以把变量保存**本地内存**（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成**数据的不一致**。

要解决这个问题，就需要把变量声明为 **volatile**，这就指示 JVM，这个变量是不稳定的，每次使用它都到主存中进行读取。说白了，**volatile** 关键字的主要作用就是保证变量的可见性然后还有一个作用是防止指令重排序。

3.2.2 说说 synchronized 关键字和 volatile 关键字的区别

synchronized 关键字和 volatile 关键字比较

volatile 关键字是线程同步的**轻量级实现**，所以 **volatile 性能肯定比 synchronized 关键字要好**。但是 **volatile 关键字只能用于变量而 synchronized 关键字可以修饰方法以及代码块**。synchronized 关键字在 JavaSE1.6 之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，**实际开发中使用 synchronized 关键字的场景还是更多一些**。

多线程访问 volatile 关键字不会发生阻塞，而 synchronized 关键字可能会发生阻塞

volatile 关键字能保证数据的可见性，但不能保证数据的原子性。

synchronized 关键字两者都能保证。

volatile 关键字主要用于解决变量在多个线程之间的可见性，而 **synchronized** 关键字解决的是多个线程之间访问资源的同步性。

3.3 线程池的 2 连击

3.3.1 为什么要用线程池？

线程池提供了一种限制和管理资源（包括执行一个任务）。每个线程池还维护一些基本统计信息，例如已完成任务的数量。

降低资源消耗。 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

提高响应速度。 当任务到达时，任务可以不需要的等到线程创建就能立即执行。

提高线程的可管理性。 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

3.3.2 实现 Runnable 接口和 Callable 接口的区别

如果想让线程池执行任务的话需要实现的 Runnable 接口或 Callable 接口。Runnable 接口或 Callable 接口实现类都可以被 ThreadPoolExecutor 或 ScheduledThreadPoolExecutor 执行。两者的区别在于 Runnable 接口不会返回结果但是 Callable 接口可以返回结果。

备注： 工具类 Executors 可以实现 Runnable 对象和 Callable 对象之间的相互转换。

（ Executors.callable（Runnable task） 或 Executors.callable（Runnable task，Object result） ）。

3.3.3 执行 execute()方法和 submit()方法的区别是什么呢？

1) execute() 方法用于提交不需要返回值的任务,所以无法判断任务是否被线程池执行成功与否；2)submit()方法用于提交需要返回值的任务。线程池会返回一个 future 类型的对象，通过这个 future 对象可以判断任务是否执行成功，并且可以通过 future 的 get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用 get (long timeout , TimeUnit unit) 方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

3.3.4 如何创建线程池

《阿里巴巴 Java 开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

Executors 返回线程池对象的弊端如下：

FixedThreadPool 和 SingleThreadExecutor ： 允许请求的队列长度为 Integer.MAX_VALUE,可能堆积大量的请求，从而导致 OOM。

CachedThreadPool 和 ScheduledThreadPool ： 允许创建的线程数量为 Integer.MAX_VALUE ，可能会创建大量线程，从而导致 OOM。

方式一：通过构造方法实现

方式二 通过 Executor 框架的工具类 Executors 来实现 我们可以创建三种类型的 ThreadPoolExecutor：

FixedThreadPool : 该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时,线程池中若有空闲线程,则立即执行。若没有,则新的任务会被暂存在一个任务队列中,待有线程空闲时,便处理在任务队列中的任务。

SingleThreadExecutor : 方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池,任务会被保存在一个任务队列中,待线程空闲,按先入先出的顺序执行队列中的任务。

CachedThreadPool : 该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定,但若有空闲线程可以复用,则会优先使用可复用的线程。若所有线程均在工作,又有新的任务提交,则会创建新的线程处理任务。所有线程在当前任务执行完毕后,将返回线程池进行复用。

对应 Executors 工具类中的方法如图所示 :

3.4 Atomic 原子类的 4 连击

3.4.1 介绍一下 Atomic 原子类

Atomic 翻译成中文是原子的意思。在化学上,我们知道原子是构成一般物质的最小单位,在化学反应中是不可分割的。在我们这里 Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候,一个操作一旦开始,就不会被其他线程干扰。所以,所谓原子类说简单点就是具有原子/原子操作特征的一类。并发包 java.util.concurrent 的原子类都存放在 java.util.concurrent.atomic 下,如下图所示。

3.4.2 JUC 包中的原子类是哪 4 类?

基本类型

使用原子的方式更新基本类型

AtomicInteger：整形原子类

AtomicLong：长整型原子类

AtomicBoolean：布尔型原子类

数组类型

使用原子的方式更新数组里的某个元素

AtomicIntegerArray：整形数组原子类

AtomicLongArray：长整形数组原子类

AtomicReferenceArray：引用类型数组原子类

引用类型

AtomicReference：引用类型原子类

AtomicStampedReference：原子更新引用类型里的字段原子类

AtomicMarkableReference：原子更新带有标记位的引用类型

对象的属性修改类型

AtomicIntegerFieldUpdater:原子更新整形字段的更新器

AtomicLongFieldUpdater：原子更新长整形字段的更新器

AtomicStampedReference：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原

子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

3.4.3 讲讲 AtomicInteger 的使用

AtomicInteger 类常用方法

AtomicInteger 类的使用示例

使用 AtomicInteger 之后，不用对 increment() 方法加锁也可以保证线程安全。

3.4.4 能不能给我简单介绍一下 AtomicInteger 类的原理

AtomicInteger 线程安全原理简单分析

AtomicInteger 类的部分源码：

```
public final int get() //获取当前的值
public final int getAndSet(int newValue) //获取当前的值，并设置新的值
public final int getAndIncrement() //获取当前的值，并自增
public final int getAndDecrement() //获取当前的值，并自减
public final int getAndAdd(int delta) //获取当前的值，并加上预期的值
boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式将该值设置为输
```

入值 (update)

```
public final void lazySet(int newValue) //最终设置为 newValue,使用 lazySet 设置之后可能导致其他线程
```

在之后的一小段时间内还是可以读到旧的值。

```
class AtomicIntegerTest {
    private AtomicInteger count = new AtomicInteger();
    //使用 AtomicInteger 之后，不需要对该方法加锁，也可以实现线程安全。
    public void increment() {
        count.incrementAndGet();
    }
    public int getCount() {
        return count.get();
    }
}
```

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS 的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。Unsafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一

个 volatile 变量,在内存中可见,因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

3.5 AQS

3.5.1 AQS 介绍

AQS 的全称为 (AbstractQueuedSynchronizer), 这个类在 java.util.concurrent.locks 包下面。

AQS 是一个用来构建锁和同步器的框架,使用 AQS 能简单且高效地构造出应用广泛的大量的同步器,比如我们提到的 ReentrantLock, Semaphore, 其他的诸如 ReentrantReadWriteLock, SynchronousQueue, FutureTask 等等皆是

基于 AQS 的。当然,我们自己也能利用 AQS 非常轻松容易地构造出符合我们自己需求的同步器。

3.5.2 AQS 原理分析

```
// setup to use Unsafe.compareAndSwapInt for updates (更新操作时提供“比较并替换”的作用)
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;
static {
    try {
        valueOffset = unsafe.objectFieldOffset
            (AtomicInteger.class.getDeclaredField("value"));
    } catch (Exception ex) { throw new Error(ex); }
}

private volatile int value;AQS 原理这部分参考了部分博客,在 5.2 节末尾放了链接。
```

在面试中被问到并发知识的时候,大多都会被问到“请你说一下自己对于 AQS 原理的理解”。下面给大家一个示例供大家参加,面试不是背题,大家一定要假如自己的思想,即使加入不了自己的思想也要保证自己能够通俗的讲出来而不是背出来。

下面大部分内容其实在 AQS 类注释上已经给出了，不过是英语看着比较吃力一点，感兴趣的话可以看看源码。

3.5.3 AQS 原理概览

AQS 核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制 AQS 是用 CLH 队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

CLH(Craig, Landin, and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS 是将每条请求共享资源的线程封装成一个 CLH 锁队列的一个结点（Node）来实现锁

的分配。看个 AQS(AbstractQueuedSynchronizer)原理图：

AQS 使用一个 int 成员变量来表示同步状态，通过内置的 FIFO 队列来完成获取资源线程的排队工作。AQS 使用 CAS 对该同步状态进行原子操作实现对其值的修改。

```
private volatile int state; // 共享变量，使用 volatile 修饰保证线程可见性
```

状态信息通过 procted 类型的 getState, setState, compareAndSetState 进行操作 // 返回同步

状态的当前值

```
protected final int getState() {  
    return state;  
}  
// 设置同步状态的值  
protected final void setState(int newState) {  
    state = newState;  
}  
// 原子地（CAS 操作）将同步状态值设置为给定值 update 如果当前同步状态的值等于 expect（期望值）  
protected final boolean compareAndSetState(int expect, int update) {  
    return unsafe.compareAndSwapInt(this, stateOffset, expect, update);  
}
```

3.5.4 AQS 对资源的共享方式

AQS 定义两种资源共享方式

Exclusive (独占) : 只有一个线程能执行, 如 `ReentrantLock`。又可分为公平锁和非公平锁:

公平锁: 按照线程在队列中的排队顺序, 先到者先拿到锁

非公平锁: 当线程要获取锁时, 无视队列顺序直接去抢锁, 谁抢到就是谁的

Share(共享) : 多个线程可同时执行, 如 `Semaphore/CountDownLatch`。

`Semaphore`。

`CountDownLatch`. `CyclicBarrier`. `ReadWriteLock` 我们都会在后面讲到。

`ReentrantReadWriteLock` 可以看成是组合式, 因为

`ReentrantReadWriteLock` 也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 `state` 的获取与释放方式即可, 至于具体线程等待队列的维护(如获取资源失败入队/唤醒出队等), `AQS` 已经在顶层实现好了。

3.5.5 AQS 底层使用了模板方法模式

同步器的设计是基于模板方法模式的, 如果需要自定义同步器一般的方式是这样(模板方法模式很经典的一个应用):

1. 使用者继承 `AbstractQueuedSynchronizer` 并重写指定的方法。(这些重写方法很简单, 无非是对于共享资源 `state` 的获取和释放)

2. 将 `AQS` 组合在自定义同步组件的实现中, 并调用其模板方法, 而这些模板方法会调用使用者重写的方法。这和我们以往通过实现接口的方式有很大区别, 这是模板方法模式很经典的一个运用。

AQS 使用了模板方法模式,自定义同步器时需要重写下面几个 AQS 提供的

模板方法：

`isHeldExclusively()`//该线程是否正在独占资源。只有用到 `condition` 才需要去实现它。

`tryAcquire(int)`//独占方式。尝试获取资源，成功则返回 `true`，失败则返回 `false`。

`tryRelease(int)`//独占方式。尝试释放资源，成功则返回 `true`，失败则返回 `false`。

`tryAcquireShared(int)`//共享方式。尝试获取资源。负数表示失败；0 表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。

`tryReleaseShared(int)`//共享方式。尝试释放资源，成功则返回 `true`，失败则返回 `false`。

默认情况下，每个方法都抛出 `UnsupportedOperationException`。这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS 类中的其他方法都是 `final`，所以无法被其他类使用，只有这几个方法可以被其他类使用。以 `ReentrantLock` 为例，`state` 初始化为 0，表示未锁定状态。A 线程 `lock()` 时，会调用 `tryAcquire()` 独占该锁并将 `state+1`。此后，其他线程再 `tryAcquire()` 时就会失败，直到 A 线程 `unlock()` 到 `state=0`（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A 线程自己是可以重复获取此锁的（`state` 会累加），这就是可重入的概念。但

要注意，获取多少次就要释放多少次，这样才能保证 `state` 是能回到零态的。

再以 `CountDownLatch` 为例，任务分为 N 个子线程去执行，`state` 也初始化为 N（注意 N 要与线程个数一致）。这 N 个子线程是并行执行的，每个子线程执行完后 `countDown()` 一次，`state` 会 `CAS(Compare and Swap)` 减 1。等到所有子线程都执行完后（即 `state=0`），会 `unpark()` 主调用线程，然后主调用线程就会从 `await()` 函数返回，继续后续动作。一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现 `tryAcquire-tryRelease`。

`tryAcquireShared-tryReleaseShared` 中的一种即可。但 AQS 也支持自定义同步器同时实现独占和共享两种方式。

如 `ReentrantReadWriteLock` 。

推荐两篇 AQS 原理和相关源码分析的文章：

<http://www.cnblogs.com/waterystone/p/4920797.html>

<https://www.cnblogs.com/chengxiao/archive/2017/07/24/7141160.html>

3.5.6 AQS 组件总结

Semaphore(信号量)-允许多个线程同时访问：`synchronized` 和 `ReentrantLock` 都是一次只允许一个线程访问

某个资源，`Semaphore(信号量)`可以指定多个线程同时访问某个资源。

CountDownLatch (倒计时器)：`CountDownLatch` 是一个同步工具类，用来协调多个线程之间的同步。这

个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。

CyclicBarrier(循环栅栏)：`CyclicBarrier` 和 `CountDownLatch` 非常类似，它也可以实现线程间的技术等待，但是它的功能比 `CountDownLatch` 更加复杂和强大。主要应用场景和 `CountDownLatch` 类似。`CyclicBarrier` 的字面意思是可循环使用 (`Cyclic`) 的屏障 (`Barrier`) 。它要做的事情是，让一组线程到达一个屏障 (也可以叫同步点) 时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。

`CyclicBarrier` 默认的构造方法是 `CyclicBarrier(int parties)` ,其参数表示屏障拦截的线程数量，每个线程调用 `await` 方法告诉 `CyclicBarrier` 我已经到达了屏障，然后当前线程被阻塞。

二. Java Web

1.JDBC 技术

1.1 说下原生 JDBC 操作数据库流程？

第一步：Class.forName()加载数据库连接驱动；

第二步：DriverManager.getConnection()获取数据连接对象；

第三步：根据 SQL 获取 sql 会话对象，有 2 种方式

Statement.PreparedStatement；

第四步：执行 SQL 处理结果集，执行 SQL 前如果有参数值就设置参数值
setXXX()；

第五步：关闭结果集.关闭会话.关闭连接。

1.2 说说事务的概念，在 JDBC 编程中处理事务的步骤。

1. 事务是作为单个逻辑工作单元执行的一系列操作。

2. 一个逻辑工作单元必须有四个属性，称为原子性.一致性.隔离性和持久性

(ACID) 属性，只有这样才能成为一个事务处理步骤：

3. conn.setAutoComit(false);设置提交方式为手工提交

4. conn.commit()提交事务

5. 出现异常，回滚 conn.rollback();

1.3 JDBC 的脏读是什么？哪种数据库隔离级别能防止脏读？

当我们使用事务时，有可能会出现这样的情况，有一行数据刚更新，与此同时另一个查询读到了这个刚更新的值。这样就导致了脏读，因为更新的数据还没

有进行持久化,更新这行数据的业务可能会进行回滚,这样这个数据就是无效的。

数据库的 `TRANSACTIONREADCOMMITTED`,

`TRANSACTIONREPEATABLEREAD` 和 `TRANSACTION_SERIALIZABLE` 隔离

级别可以防止脏读

2.网路通讯部分

2.1 TCP 与 UDP 区别 ?

UDP:

- a.是面向无连接, 将数据及源的封装成数据包中,不需要建立连接
- b.每个数据报的大小在限制 64k 内
- c.因无连接,是不可靠协议
- d.不需要建立连接,速度快

TCP :

- a.建立连接,形成传输数据的通道.
- b.在连接中进行大数据量传输, 以字节流方式
- c.通过三次握手完成连接,是可靠协议
- d.必须建立连接效率会稍低.聊天.网络视频会议就是 UDP

2.2 说一下什么是 Http 协议 ?

客户端和 服务器端之间数据传输的格式规范, 格式简称为“超文本传输协议”。 是一个基于请求与响应模式的.无状态的.应用层的协议, 基于 TCP 的连接方式

2.3 get 与 post 请求区别 ?

区别 1:

get 重点在从服务器上获取资源, post 重点在向服务器发送数据;

区别 2 :

get 传输数据是通过 URL 请求, 以 field (字段) = value 的形式, 置于 URL 后, 并用"?"连接, 多个请求数据间用"&"连接, 如
`http://127.0.0.1/Test/login.action?name=admin&password=admin`, 这个过程用户是可见的;

post 传输数据通过 Http 的 post 机制，将字段与对应值封存在请求实体中发送给服务器，这个过程对用户是不可见的；

区别 3：

Get 传输的数据量小，因为受 URL 长度限制，但效率较高；

Post 可以传输大量数据，所以上传文件时只能用 Post 方式；

区别 4：

Get 是不安全的，因为 URL 是可见的，可能会泄露私密信息，如密码等；

Post 较 get 安全性较高；

区别 5：

get 方式只能支持 ASCII 字符，向服务器传的中文字符可能会乱码。

post 支持标准字符集，可以正确传递中文字符。

2.4 http 中重定向和请求转发的区别？

本质区别：转发是服务器行为，重定向是客户端行为。

重定向特点：两次请求，浏览器地址发生变化，可以访问自己 web 之外的资源，传输的数据会丢失。

请求转发特点：一次强求，浏览器地址不变，访问的是自己本身的 web 资源，传输的数据不会丢失。

3. Cookie 和 Session

Cookie 是 web 服务器发送给浏览器的一块信息，浏览器会在本地一个文件中给每个 web 服务器存储 cookie。以后浏览器再给特定的 web 服务器发送请求时，同时会发送所有为该服务器存储的 cookie。

Session 是存储在 web 服务器端的一块信息。session 对象存储特定用户会话所需的属性及配置信息。当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。

Cookie 和 session 的不同点：

1.无论客户端做怎样的设置，session 都能够正常工作。当客户端禁用 cookie 时将无法使用 cookie。

2.在存储的数据量方面：session 能够存储任意的 java 对象，cookie 只能存储 String 类型的对象。

4.Jsp 和 Servlet

4.1 Servlet 的执行流程

Servlet 的执行流程也就是 servlet 的生命周期，当服务器启动的时候生命周期开始，然后通过 init()《启动顺序根据 web.xml 里的 startup-on-load 来确定加载顺序》方法初始化 servlet，再根据不同请求调用 doGet 或 doPost 方法，最后再通过 destroy()方法进行销毁。

4.2 doGet 和 doPost 的区别

doGet 和 doPost 都是接受用户请求的方法，doGet 处理 get 请求，doPost 处理 post 请求，doGet 用于地址栏提交，doPost 用于表单提交，在页面提交数据时，get 的数据大小有限制 4k，post 没有限制，get 请求提交的数据会在地址栏显示，post 不显示，所以 post 比 get 安全。

4.3 Jsp 和 Servlet 的区别

你可以将 JSP 当做一个可扩充的 HTML 来对待。

虽然在本质上 JSP 文件会被服务器自动翻译为相应的 Servlet 来执行。

可以说 Servlet 是面向 Java 程序员而 JSP 是面向 HTML 程序员的，除此之外两者功能完全等价。

4.4 JSP 九大内置对象

pageContext：只对当前 jsp 页面有效，里面封装了基本的 request 和 session 的对象

Request : 对当前请求进行封装

Session : 浏览器会话对象, 浏览器范围内有效

Application : 应用程序对象, 对整个 web 工程都有效

Out : 页面打印对象, 在 jsp 页面打印字符串

Response : 返回服务器端信息给用户

Config : 单个 servlet 的配置对象, 相当于 servletConfig 对象

Page : 当前页面对象, 也就是 this

Exception : 错误页面的 exception 对象, 如果指定的是错误页面, 这个就是异常对象

4.5 JSP 的三大指令 :

Page : 指令是针对当前页面的指令

Include : 用于指定如何包含另一个页面

Taglib : 用于定义和指定自定义标签

4.6 七大动作

Forward , 执行页面跳转, 将请求的处理转发到另一个页面

Param : 用于传递参数

Include : 用于动态引入一个 jsp 页面

Plugin : 用于下载 javaBean 或 applet 到客户端执行

useBean : 使用 javaBean

setProperty : 修改 javaBean 实例的属性值

getProperty : 获取 javaBean 实例的属性值

5. Ajax &Jquery

5.1 谈谈你对 Ajax 的认识 ?

Ajax 是一种创建交互式网页应用的的网页开发技术 ; Asynchronous

JavaScript and XML” 的缩写。

Ajax 的优势 :

通过异步模式，提升了用户体验。

优化了浏览器和服务端之间的传输，减少不必要的数据往返，减少了带宽占用。

Ajax 引擎在客户端运行，承担了一部分本来由服务器承担的工作，从而减少了大用户量下的服务器负载。

Ajax 的最大特点：

可以实现局部刷新，在不更新整个页面的前提下维护数据，提升用户体验度。

5.2 使用 JQuery 手写 Ajax

```
$.ajax({  
    url:'http://www.baidu.com',  
    type:'POST',  
    data:data,  
    cache:true,  
    headers:{},  
    beforeSend : function(){},  
    success:function(){},  
    error:function(){},  
    complete:function(){  
    }  
});
```

5.3 请简单介绍 Ajax 的使用

Ajax = 异步 JavaScript 和 XML。

Ajax 是一种用于创建快速动态网页的技术。

通过在后台与服务器进行少量数据交换，AJAX 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。

传统的网页（不使用 AJAX）如果需要更新内容，必需重载整个网页面。

有很多使用 AJAX 的应用程序案例：新浪微博、Google 地图、开心网等等。

5.4 Ajax 可以做异步请求么？

可以。ajax 请求默认是异步的。如果想同步，把 `async` 设置为 `false` 就可以了。默认是 `true`。

如果是 jquery

```
$.ajax({  
  url: some.php,  
  async: false,  
  success : function(){  
  }  
});
```

如果是原生的 js

```
xmlHttp.open("POST",url,false);
```

5.5 请介绍下 Jsonp 原理

jsonp 的最基本的原理是：动态添加一个 `<script>` 标签，使用 `script` 标签的 `src` 属性没有跨域的限制的特点实现跨域。首先在客户端注册一个 `callback`，然后把 `callback` 的名字传给服务器。此时，服务器先生成 json 数据。然后以 javascript 语法的方式，生成一个 `function`，`function` 名字就是传递上来的参数 `jsonp`。最后将 json 数据直接以入参的方式，放置到 `function` 中，这样就生成了一段 js 语法的文档，返回给客户端。客户端浏览器，解析 `script` 标签，并执行返回的 javascript 文档，此时数据作为参数，传入到了客户端预先定义好的 `callback` 函数里。

5.6 JQuery 常用选择器

1.ID 选择器 #id

描述：根据给定的 id 匹配一个元素，返回单个元素（注：在网页中，id 名称不能重复）

示例：\$("#test") 选取 id 为 test 的元素

2.类选择器 .class

描述：根据给定的类名匹配元素，返回元素集合

示例：\$(".test") 选取所有 class 为 test 的元素

3.元素选择器 element

描述：根据给定的元素名匹配元素，返回元素集合

示例：\$("p") 选取所有的<p>元素

4. *

描述：匹配所有元素，返回元素集合

示例：\$("*") 选取所有的元素

三. 数据库

3.1 SQL 之连接查询

外连接：

1) 左连接 (左外连接) 以左表为基准进行查询,左表数据会全部显示出来,右表 如果和左表匹配 的数据则显示相应字段的数据,如果不匹配,则显示为 NULL;

2) 右连接 (右外连接) 以右表为基准进行查询,右表数据会全部显示出来,右表 如果和左表匹配的数据则显示相应字段的数据,如果不匹配,则显示为

NULL;

3) 全连接就是先以左表进行左外连接, 然后以右表进行右外连接。

内连接:

显示表之间有连接匹配的所有行。

3.2 SQL 之聚合函数

聚合函数是对一组值执行计算并返回单一的值的函数, 它经常与 SELECT 语句的 GROUP BY 子句一同使用。

1) .AVG 返回指定组中的平均值, 空值被忽略; 例: `select prd_no, avg(qty) from sales group by prd_no`

2) COUNT 返回指定组中 项目的数量。 例: `select count(*) from sales;`

2). MAX 返回指定数据的最大值; MIN 返回指定数据的最小值; SUM 返回指定数据的和, 只能用于数字列, 空值被忽略。 例: `select prd_no, max(qty) from sales group by prd_no`

3) 使用 group by 子句对数据进行分组; 对 group by 子句形成的组运行 聚集函数计算每一组的值; 最后用 having 子句去掉不符合条件的组; having 子 句中的每一个元素也必须出现在 select 列表中。有些数据库例外, 如 oracle. 例: `select prd_no, max(qty) from sales group by prd_no having prd_no > 10`

3.3 SQL 之 SQL 注入

举例:

`select admin from user where username='admin' or 'a'='a' and passwd='' or 'a'='a'`

防止 SQL 注入，使用预编译语句是预防 SQL 注入的最佳方式，如

```
select admin from user where username= ? And password=?
```

使用预编译的 SQL 语句语义不会发生改变，在 SQL 语句中，变量用问号？表示。像上面例子中，username 变量传递的 'admin' or 'a'='a' 参数，也只会当作 username 字符串来解释查询，从根本上杜绝了 SQL 注入攻击的发生。

注意：使用 mybaits 时 mapper 中 # 方式能够很大程度防止 SQL 注入，\$ 方式无法防止 SQL 注入。

3.4 SQL Select 语句完整的执行顺序:

查询中用到的关键词主要包含六个，并且他们的顺序依次为

```
select--from--where--group by--having--order by
```

其中 select 和 from 是必须的，其他关键词是可选的，

这六个关键词的执行顺序如下：

from: 需要从哪个数据表检索数据

where: 过滤表中数据的条件

group by: 如何将上面过滤出的数据分组

having: 对上面已经分组的数据进行过滤的条件

select: 查看结果集中的哪个列，或列的计算结果

order by : 按照什么样的顺序来查看返回的数据

3.5 存储引擎

1. 概念

数据库存储引擎是数据库底层软件组织，数据库管理系统 (DBMS) 使用数

据引擎进行创建、查询、更新和删除数据。不同的存储引擎提供不同的存储机制。

索引技巧、锁定水平等功能，使用不同的存储引擎，还可以获得特定的功能。

现在许多不同的数据库管理系统都支持多种不同的数据引擎。

存储引擎主要有：1. MyIsam, 2. InnoDB, 3. Memory, 4. Archive, 5. Federated。

2. InnoDB

InnoDB 底层存储结构为 B+树，B树的每个节点对应 innodb 的一个 page，page 大小是固定的，一般设为 16k。其中非叶子节点只有键值，叶子节点包含完整数据。

适用场景：

- 1) 经常更新的表，适合处理多重并发的更新请求。
- 2) 支持事务。
- 3) 可以从灾难中恢复（通过 bin-log 日志等）。
- 4) 外键约束。只有他支持外键。
- 5) 支持自动增加列属性 auto_increment。

3. TokuDB

TokuDB 底层存储结构为 Fractal Tree, Fractal Tree 的结构与 B+树有些类似，在 Fractal Tree 中，每一个 child 指针除了需要指向一个 child 节点外，还会带有一个 Message Buffer，这个 Message Buffer 是一个 FIFO 的队列，用来缓存更新操作。

例如，一次插入操作只需要落在某节点的 Message Buffer 就可以马上返回了，并不需要搜索到叶子节点。这些缓存的更新会在查询时或后台异步合并

应用到对应的节点中。

TokuDB 在线添加索引，不影响读写操作，非常快的写入性能，Fractal-tree 在事务实现上有优势。他主要适用于访问频率不高的数据或历史数据归档。

4. MyIASM

MyIASM 是 MySQL 默认的引擎，但是它没有提供对数据库事务的支持，也不支持行级锁和外键，因此当 INSERT(插入)或 UPDATE(更新)数据时即写操作需要锁定整个表，效率便会低一些。

ISAM 执行读取操作的速度很快，而且不占用大量的内存和存储资源。在设计之初就预想数据组织成有固定长度的记录，按顺序存储的。---ISAM 是一种静态索引结构。缺点是它不支持事务处理。

5. Memory

Memory (也叫 HEAP) 堆内存：使用存在内存中的内容来创建表。每个 MEMORY 表只实际对应一个磁盘文件。MEMORY 类型的表访问非常得快，因为它的数据是放在内存中的，并且默认使用 HASH 索引。但是一旦服务关闭，表中的数据就会丢失掉。Memory 同时支持散列索引和 B 树索引，B 树索引可以使用部分查询和通配查询，也可以使用和 >= 等操作符方便数据挖掘，散列索引相等的比较快但是对于范围的比较慢很多。

3.6 索引

索引 (Index) 是帮助 MySQL 高效获取数据的数据结构。常见的查询算法，顺序查找，二分查找，二叉排序树查找，哈希散列法，分块查找，平衡多路搜索树 B 树 (B-tree)

索引就是加快检索表中数据的方法。数据库的索引类似于书籍的索引。在书籍中,索引允许用户不必翻阅完整本书就能迅速地找到所需要的信息。在数据库中,索引也允许数据库程序迅速地找到表中的数据,而不必扫描整个数据库。

MySQL 数据库几个基本的索引类型:普通索引.唯一索引.主键索引.全文索引.组合索引

1.普通索引

是最基本的索引,它没有任何限制。它有以下几种创建方式:

(1) 直接创建索引

```
1 CREATE INDEX index_name ON table(column[length])
```

(2) 修改表结构的方式添加索引

```
1 ALTER TABLE table_name ADD INDEX index_name ON (column[length])
```

(3) 创建表的时候同时创建索引

```
1 CREATE TABLE `table` (  
2     `id` int(11) NOT NULL AUTO_INCREMENT ,  
3     `title` char(255) CHARACTER NOT NULL ,  
4     `content` text CHARACTER NULL ,  
5     `time` int(10) NULL DEFAULT NULL ,  
6     PRIMARY KEY (`id`),  
7     INDEX index_name (title[length])  
8 )
```

(4) 删除索引

1	DROP INDEX index_name ON table
---	--------------------------------

2.唯一索引

与前面的普通索引类似,不同的就是:索引列的值必须唯一,但允许有空值。

如果是组合索引,则列值的组合必须唯一。它有以下几种创建方式:

(1) 创建唯一索引

1	CREATE UNIQUE INDEX indexName ON table(column[length])
---	--

(2) 修改表结构

1	ALTER TABLE table_name ADD UNIQUE indexName ON (column[length])
---	---

(3) 创建表的时候直接指定

1	CREATE TABLE `table` (
2	`id` int(11) NOT NULL AUTO_INCREMENT ,
3	`title` char(255) CHARACTER NOT NULL ,
4	`content` text CHARACTER NULL ,
5	`time` int(10) NULL DEFAULT NULL ,
6	UNIQUE indexName (title[length])
7	);

3.主键索引

是一种特殊的唯一索引,一个表只能有一个主键,不允许有空值。一般是在建表的时候同时创建主键索引:

1	CREATE TABLE `table` (
---	------------------------

2	<code>`id` int(11) NOT NULL AUTO_INCREMENT ,</code>
3	<code>`title` char(255) NOT NULL ,</code>
4	<code>PRIMARY KEY (`id`)</code>
5	<code>);</code>

4.组合索引

指多个字段上创建的索引 ,只有在查询条件中使用了创建索引时的第一个字段 ,索引才会被使用。使用组合索引时遵循最左前缀集合

1	<code>ALTER TABLE `table` ADD INDEX name_city_age (name,city,age);</code>
---	---

5.全文索引

主要用来查找文本中的关键字 ,而不是直接与索引中的值相比较。fulltext 索引跟其它索引大不相同 ,它更像是一个搜索引擎 ,而不是简单的 where 语句的参数匹配。fulltext 索引配合 match against 操作使用 ,而不是一般的 where 语句加 like。它可以在 create table , alter table , create index 使用 ,不过目前只有 char varchar , text 列上可以创建全文索引。值得一提的是 ,在数据量较大时候 ,现将数据放入一个没有全局索引的表中 ,然后再用 CREATE index 创建 fulltext 索引 ,要比先为一张表建立 fulltext 然后再将数据写入的速度快很多。

(1) 创建表的适合添加全文索引

1	<code>CREATE TABLE `table` (</code>
2	<code>`id` int(11) NOT NULL AUTO_INCREMENT ,</code>
3	<code>`title` char(255) CHARACTER NOT NULL ,</code>

4	<code>`content` text CHARACTER NULL ,</code>
5	<code>`time` int(10) NULL DEFAULT NULL ,</code>
6	<code>PRIMARY KEY (`id`),</code>
7	<code>FULLTEXT (content)</code>
8	<code>);</code>

(2) 修改表结构添加全文索引

1	<code>ALTER TABLE article ADD FULLTEXT index_content(content)</code>
---	--

(3) 直接创建全文索引

1	<code>CREATE FULLTEXT INDEX index_content ON article(content)</code>
---	--

索引的优点

创建唯一性索引，保证数据库表中每一行数据的唯一性

大大加快数据的检索速度，这也是创建索引的最主要的原因

加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

通过使用索引，可以在查询的过程中使用优化隐藏器，提高系统的性能。

索引的缺点

创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加

索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大

当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，降低了数据的维护速度

常见索引原则有

选择唯一性索引：唯一性索引的值是唯一的，可以更快速的通过该索引来确定某条记录。

为经常需要排序、分组和联合操作的字段建立索引。

为常作为查询条件的字段建立索引。

限制索引的数目：越多的索引，会使更新表变得很浪费时间。

尽量使用数据量少的索引：如果索引的值很长，那么查询的速度会受到影响。

尽量使用前缀来索引：如果索引字段的值很长，最好使用值的前缀来索引。

删除不再使用或者很少使用的索引

最左前缀匹配原则，非常重要的原则。

尽量选择区分度高的列作为索引：区分度的公式是表示字段不重复的比例

索引列不能参与计算，保持列“干净”：带函数的查询不参与索引。

尽量扩展索引，不要新建索引。

3.7 数据库三范式

范式是具有最小冗余的表结构。3 范式具体如下：

1. 第一范式(1st NF - First Normal Formate)

第一范式的目标是确保每列的原子性：如果每列都是不可再分的最小数据单元（也称为最小的原子单元），则满足第一范式（1NF）

第一范式（1NF）要求数据库表的每一列都是不可分割的基本数据项，同一列中不能有多值。

若某一列有多个值,可以将该列单独拆分成一个实体,新实体和原实体间是一对多的关系。

在任何一个关系数据库中,第一范式(1NF)是对关系模式的基本要求,不满足第一范式(1NF)的数据库就不是关系数据库。

第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值,就说明该数据库表满足了第一范式。

第一范式的合理遵循需要根据系统的实际需求来定。比如某些数据库系统中需要用到“地址”这个属性,本来直接将“地址”属性设计成一个数据库表的字段就行。但是如果系统经常会访问“地址”属性中的“城市”部分,那么就非要将“地址”这个属性重新拆分为省份.城市.详细地址等多个部分进行存储,这样在对地址中某一部分操作的时候将非常方便。这样设计才算满足了数据库的第一范式

2. 第二范式(2nd NF - Second Normal Fromate)

首先满足第一范式,并且表中非主键列不存在对主键的部分依赖。第二范式要求每个表只描述一 件事情。

满足第二范式(2NF)必须先满足第一范式(1NF)。

第二范式要求实体中没一行的所有非主属性都必须完全依赖于主键;即:非主属性必须完全依赖于主键。

完全依赖:主键可能由多个属性构成,完全依赖要求不允许存在非主属性依赖于主键中的某一部分属性。

若存在哪个非主属性依赖于主键中的一部分属性,那么要将发生部分依赖的这一组属性单独新建一个实体,并且在旧实体中用外键与新实体关联,并且新实

体与旧实体间是一对多的关系。

第二范式在第一范式的基础之上更进一层。第二范式需要确保数据库表中的每一列都和主键相关,而不能只与主键的某一部分相关(主要针对联合主键而言)。也就是说在一个数据库表中,一个表中只能保存一种数据,不可以把多种数据保存在同一张数据库表中。

3. 第三范式(3rd NF - Third Normal Fromate)

第三范式定义是,满足第二范式,并且表中的列不存在对非主键列的传递依赖。除了主键订单编号外,顾客姓名依赖于非主键顾客编号。

满足第三范式必须先满足第二范式。

第三范式要求:实体中的属性不能是其他实体中的非主属性。因为这样会出现冗余。即:属性不依赖于其他非主属性。

如果一个实体中出现其他实体的非主属性,可以将这两个实体用外键关联,而不是将另一张表的非主属性直接写在当前表中。

第三范式需要确保数据表中的每一列数据都和主键直接相关,而不能间接相关。

3.8 数据库事务

1. 事务(Transaction)是作为单个逻辑工作单元执行的一系列操作,这些操作作为一个整体一起向系统提交,要么都执行,要么都不执行。事务是一个不可分割的工作逻辑单元,事务必须具备以下四个属性,简称 ACID 属性:

A 原子性 (Atomicity):事务是一个完整的操作。事务的各步操作是不可分的(原子的);要么都执行,要么都不执行。

B 一致性 (Consistency) : 当事务完成时 , 数据必须处于一致状态。

C 隔离性 (Isolation) : 对数据进行修改的所有并发事务是彼此隔离的 , 这表明事务必须是独立的 , 它不应以任何方式依赖于或影响其他事务。

D 永久性 (Durability) : 事务完成后 , 它对数据库的修改被永久保持 , 事务日志能够保持事务 的永久性。

2. 事务控制语句 :

- BEGIN 或 START TRANSACTION 显式地开启一个事务 ;
- COMMIT 也可以使用 COMMIT WORK , 不过二者是等价的。
COMMIT 会提交事务 , 并使已对数据库进行的所有修改成为永久性的 ;
- ROLLBACK 也可以使用 ROLLBACK WORK , 不过二者是等价的。
回滚会结束用户的事务 , 并撤销正在进行的所有未提交的修改 ;
- SAVEPOINT identifier , SAVEPOINT 允许在事务中创建一个保存点 , 一个事务中可以有多个 SAVEPOINT ;
- RELEASE SAVEPOINT identifier 删除一个事务的保存点 , 当没有指定的保存点时 , 执行该语句会抛出一个异常 ;
- ROLLBACK TO identifier 把事务回滚到标记点 ;
- SET TRANSACTION 用来设置事务的隔离级别。InnoDB 存储引擎提供事务的隔离级别有 READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ 和 SERIALIZABLE。

3. MySQL 事务处理主要有两种方法 :

1) 用 BEGIN, ROLLBACK, COMMIT 来实现

a) BEGIN 开始一个事务

b) ROLLBACK 事务回滚

c) COMMIT 事务确认

2) 直接用 SET 来改变 MySQL 的自动提交模式:

a) SET AUTOCOMMIT=0 禁止自动提交

b) SET AUTOCOMMIT=1 开启自动提交

4. 事务的四种隔离级别

1) Read uncommitted

读未提交,顾名思义,就是一个事务可以读取另一个未提交事务的数据。

2) Read committed

读已提交,顾名思义,就是一个事务要等另一个事务提交后才能读取数据。

3) Repeatable read

可重复读,就是在开始读取数据(事务开启)时,不再允许修改操作

4) Serializable 序列化

Serializable 是最高的事务隔离级别,在该级别下,事务串行化顺序执行,可以避免脏读、不可重复读与幻读。但是这种事务隔离级别效率低下,比较耗数据库性能,一般不使用。

在 MySQL 数据库中,支持上面四种隔离级别,默认的为 Repeatable read (可重复读);而在 Oracle 数据库中,只支持 Serializable (串行化)级别和 Read committed (读已提交)这两种级别,其中默认的为 Read committed 级别。

3.9 存储过程

一组为了完成特定功能的 SQL 语句集，存储在数据库中，经过第一次编译后再次调用不需要再次编译，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。存储过程是数据库中的一个重要对象。

存储过程优化思路：

1. 尽量利用一些 SQL 语句来替代一些小循环，例如聚合函数，求平均函数等。
2. 中间结果存放于临时表，加索引。
3. 少使用游标。SQL 是个集合语言，对于集合运算具有较高性能。而 cursors 是过程运算。比如对一个 100 万行的数据进行查询。游标需要读表 100 万次，而不使用游标则只需要少量几次读取。
4. 事务越短越好。SQLserver 支持并发操作。如果事务过多过长，或者隔离级别过高，都会造成并发操作的阻塞，死锁。导致查询极慢，cpu 占用率极高。
5. 使用 try-catch 处理错误异常。
6. 查找语句尽量不要放在循环内。

3.10 触发器

触发器是一段能自动执行的程序，是一种特殊的存储过程，触发器和普通的存储过程的区别是：触发器是当对某一个表进行操作时触发。诸如：update.insert.delete 这些操作的时候，系统会自动调用执行该表上对应的触发器。SQL Server 2005 中触发器可以分为两类：DML 触发器和 DDL 触发器，其中 DDL 触发器它们会影响多种数据定义语言语句而激发，这些语句有

create. alter.drop 语句。

3.11 数据库并发策略

并发控制一般采用三种方法，分别是乐观锁和悲观锁以及时间戳。

乐观锁

乐观锁认为一个用户读数据的时候，别人不会去写自己所读的数据；悲观锁就刚好相反，觉得自己读数据库的时候，别人可能刚好在写自己刚读的数据，其实就是持一种比较保守的态度；时间戳就是不加锁，通过时间戳来控制并发出现的问题。

悲观锁

悲观锁就是在读取数据的时候，为了不让别人修改自己读取的数据，就会先对自己读取的数据加锁，只有自己把数据读完了，才允许别人修改那部分数据，或者反过来说，就是自己修改某条数据的时候，不允许别人读取该数据，只有等自己的整个事务提交了，才释放自己加上的锁，才允许其他用户访问那部分数据。

两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行 retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

乐观锁常见的两种实现方式

版本号机制

一般是在数据表中加上一个数据版本号 version 字段,表示数据被修改的次数,当数据被修改时,version 值会加一。当线程 A 要更新数据值时,在读取数据的同时也会读取 version 值,在提交更新时,若刚才读取到的 version 值为当前数据库中的 version 值相等时才更新,否则重试更新操作,直到更新成功。

CAS 算法即 compare and swap (比较与交换), 是一种有名的无锁算法。无锁编程,即不使用锁的情况下实现多线程之间的变量同步,也就是在没有线程被阻塞的情况下实现变量的同步,所以也叫非阻塞同步 (Non-blocking Synchronization)。CAS 算法涉及到三个操作数

需要读写的内存值 V

进行比较的值 A

拟写入的新值 B

当且仅当 V 的值等于 A 时, CAS 通过原子方式用新值 B 来更新 V 的值,否则不会执行任何操作 (比较和替换是一个原子操作)。一般情况下是一个自旋操作,即不断的重试。

乐观锁的缺点

ABA 问题

如果一个变量 V 初次读取的时候是 A 值,并且在准备赋值的时候检查到它仍然是 A 值,那我们就能说明它的值没有被其他线程修改过了吗?很明显是不能的,因为在这段时间它的值可能被改为其他值,然后又改回 A,那 CAS 操作就会误认为它从来没有被修改过。这个问题被称为 CAS 操作的 "ABA"问题。

JDK 1.5 以后的 AtomicStampedReference 类就提供了此种能力,其中的 compareAndSet 方法就是首先检查当前引用是否等于预期引用,并且当前

标志是否等于预期标志,如果全部相等,则以原子方式将该引用和该标志的值设置为给定的更新值。

循环时间长开销大

自旋 CAS (也就是不成功就一直循环执行直到成功) 如果长时间不成功, 会给 CPU 带来非常大的执行开销。如果 JVM 能支持处理器提供的 pause 指令那么效率会有一定的提升, pause 指令有两个作用, 第一它可以延迟流水线执行指令 (de-pipeline), 使 CPU 不会消耗过多的执行资源, 延迟的时间取决于具体实现的版本, 在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突(memory order violation)而引起 CPU 流水线被清空(CPU pipeline flush), 从而提高 CPU 的执行效率。

只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效, 当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5 开始, 提供了 AtomicReference 类来保证引用对象之间的原子性, 你可以把多个变量放在一个对象里来进行 CAS 操作. 所以我们可以使用锁或者利用 AtomicReference 类把多个共享变量合并成一个共享变量来操作

CAS 与 synchronized 的使用情景

简单的来说 CAS 适用于写比较少的情况下 (多读场景, 冲突一般较少), synchronized 适用于写比较多的情况下 (多写场景, 冲突一般较多)

对于资源竞争较少 (线程冲突较轻) 的情况, 使用 synchronized 同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗 cpu 资源; 而 CAS 基于硬件实现, 不需要进入内核, 不需要切换线程, 操作自旋几率较少, 因此可以获得更高的性能。

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

补充：Java 并发编程这个领域中 synchronized 关键字一直都是元老级的角色，很久之前很多人都会称它为“重量级锁”。但是，在 JavaSE 1.6 之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后变得在某些情况下并不是那么重了。

synchronized 的底层实现主要依靠 Lock-Free 的队列，基本思路是自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和 CAS 类似的性能；而线程冲突严重的情况下，性能远高于 CAS。

时间戳

时间戳就是在数据库表中单独加一列时间戳，比如“TimeStamp”，每次读出来的时候，把该字段也读出来，当写回去的时候，把该字段加 1，提交之前，跟数据库的该字段比较一次，如果比数据库的值大的话，就允许保存，否则不允许保存，这种处理方法虽然不使用数据库系统提供的锁机制，但是这种方法可以大大提高数据库处理的并发量，

以上悲观锁所说的加“锁”，其实分为几种锁，分别是：排它锁（写锁）和共享锁（读锁）。

3.12 数据库锁

1. 行级锁

行级锁是一种排他锁，防止其他事务修改此行；在使用以下语句时，Oracle 会自动应用行级锁：

INSERT.UPDATE.DELETE.SELECT ... FOR UPDATE [OF columns] [WAIT
n | NOWAIT];

SELECT ... FOR UPDATE 语句允许用户一次锁定多条记录进行更新
使用 COMMIT 或 ROLLBACK 语句释放锁。

2. 表级锁

表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分
MySQL 引擎支持。最常使用的 MYISAM 与 INNODB 都支持表级锁定。表
级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）。

3. 页级锁

页级锁是 MySQL 中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁
速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定
相邻的一组记录。BDB 支持页级锁

3.13 基于 Redis 分布式锁

1. 获取锁的时候，使用 setnx (ETNX key val：当且仅当 key 不存在时，
set 一个 key 为 val 的字符串，返回 1；若 key 存在，则什么都不做，返回 0)
加锁，锁的 value 值为一个随机生成的 UUID，在释放锁的时候进行判断。并
使用 expire 命令为锁添加一个超时时间，超过该时间则自动释放锁。

2. 获取锁的时候调用 setnx，如果返回 0，则该锁正在被别人使用，返回 1
则成功获取锁。还设置一个获取的超时时间，若超过这个时间则放弃获取锁。

3. 释放锁的时候，通过 UUID 判断是不是该锁，若是该锁，则执行 delete
进行锁释放。

3.14 分区分表

分库分表有垂直切分和水平切分两种。

- 垂直切分：将表按照功能模块.关系密切程度划分出来，部署到不同的库上。例如，我们会 建立定义数据库 workDB.商品数据库 payDB.用户数据库 userDB.日志数据库 logDB 等，分别用于存储项目数据定义表.商品定义表.用户数据表.日志数据表等。

- 水平切分：当一个表中的数据量过大时，我们可以把该表的数据按照某种规则，例如 userID 散列，进行划分，然后存储到多个结构相同的表，和不同的库上。例如，我们的 userDB 中的用户数据表中，每一个表的数据量都很大，就可以把 userDB 切分为结构相同的多个 userDB : part0DB.part1DB 等，再将 userDB 上的用户数据表 userTable，切分为很多 userTable : userTable0.userTable1 等，然后将这些表按照一定的规则存储到多个 userDB 上。

3.15 应该使用哪一种方式来实施数据库分库分表，这要看数据库中数据量的瓶颈 所在，并综合项目的业务类型进行考虑。

如果数据库是因为表太多而造成海量数据，并且项目的各项业务逻辑划分清晰.低耦合，那么规则简单明了.容易实施的垂直切分必是首选。

而如果数据库中的表并不多，但单表的数据量很大.或数据热度很高，这种情况 之下就应该选择水平切分，水平切分比垂直切分要复杂一些，它将原本逻辑上属 于一体的数据进行了物理分割，除了在分割时要对分割的粒度做好评估，考虑数 据平均和负载平均，后期也将对项目人员及应用程序产生额外的数据管

理负担。在现实项目中，往往是这两种情况兼而有之，这就需要做出权衡，甚至既需要垂直切分，又需要水平切分。我们的游戏项目便综合使用了垂直与水平切分，我们首先对数据库进行垂直切分，然后，再针对一部分表，通常是用户数据表，进行水平切分。

单库多表：

随着用户数量的增加，user 表的数据量会越来越大，当数据量达到一定程度的时候对 user 表的查询会渐渐的变慢，从而影响整个 DB 的性能。如果使用 MySQL，还有一个更严重的问题是，当需要添加一列的时候，MySQL 会锁表，期间所有的读写操作只能等待。

可以将 user 进行水平的切分，产生两个表结构完全一样的 user_0000, user_0001 等表，user_0000 + user_0001 + ... 的数据刚好是一份完整的数据。

多库多表：

随着数据量增加也许单台 DB 的存储空间不够，随着查询量的增加单台数据库服务器已经没办法支撑。这个时候可以再对数据库进行水平区分。

分库分表规则举例：通过分库分表规则查找到对应的表和库的过程。如分库分表的规则是 user_id 除以 4 的方式，当用户新注册了一个账号，账号 id 的 123，我们可以通过 id 除以 4 的方式确定此账号应该保存到 User_0003 表中。当用户 123 登录的时候，我们通过 123 除以 4 后确定记录在 User_0003 中。

3.16 MySQL 读写分离

在实际的应用中，绝大部分情况都是读远大于写。MySQL 提供了读写分离

的机制,所有的写操作都必须对应到 Master,读操作可以在 Master 和 Slave 机器上进行,Slave 与 Master 的结构完全一样,一个 Master 可以有多个 Slave,甚至 Slave 下还可以挂 Slave,通过此方式可以有效的提高 DB 集群的每秒查询率. 所有的写操作都是先在 Master 上操作,然后同步更新到 Slave 上,所以从 Master 同步到 Slave 机器有一定的延迟,当系统很繁忙的时候,延迟问题会更加严重,Slave 机器数量的增加也会使这个问题更加严重。

此外,可以看出 Master 是集群的瓶颈,当写操作过多,会严重影响到 Master 的稳定性,如果 Master 挂掉,整个集群都将不能正常工作。所以,

1. 当读压力很大的时候,可以考虑添加 Slave 机器的分式解决,但是当 Slave 机器达到一定的数量就得考虑分库了。
2. 当写压力很大的时候,就必须得进行分库操作。

3.17 MySQL 常用 30 种 SQL 查询语句优化方法

- 1.应尽量避免在 where 子句中使用!=或<>操作符,否则引擎将放弃使用索引而进行全表扫描。

- 2.对查询进行优化,应尽量避免全表扫描,首先应考虑在 where 及 order by 涉及的列上建立索引。

- 3.应尽量避免在 where 子句中对字段进行 null 值判断,否则将导致引擎放弃使用索引而进行全表扫描。如:

```
select id from t where num is null
```

可以在 num 上设置默认值 0,确保表中 num 列没有 null 值,然后这样查询:

```
select id from t where num=0
```

4.尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使

用索引而进行全表扫描，如：

```
select id from t where num=10 or num=20
```

可以这样查询：

```
select id from t where num=10
```

```
union all
```

```
select id from t where num=20
```

5.下面的查询也将导致全表扫描：(不能前置百分号)

```
select id from t where name like 'c%
```

下面走索引

```
select id from t where name like 'c%'
```

若要提高效率，可以考虑全文检索。

6.in 和 not in 也要慎用，否则会导致全表扫描，如：

```
select id from t where num in(1,2,3)
```

对于连续的数值，能用 between 就不要用 in 了：select id from t where num between 1 and 3

7.如果在 where 子句中使用参数，也会导致全表扫描。因为 SQL 只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

```
select id from t where num=@num
```

可以改为强制查询使用索引：

```
select id from t with(index(索引|名)) where num=@num
```

8.应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
select id from t where num/2=100
```

应改为:

```
select id from t where num=100*2
```

9.应尽量避免在 where 子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
select id from t where substring(name,1,3)=' abc' --name 以 abc 开头的 id
```

```
select id from t where datediff(day,createdate,' 2005-11-30')=0 --' 2005-11-30'生成的 id
```

应改为:

```
select id from t where name like 'abc%'
```

```
select id from t where createdate>=' 2005-11-30' and createdate<' 2005-12-1'
```

10.不要在 where 子句中的 “=” 左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

11.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

12.不要写一些没有意义的查询，如需要生成一个空表结构：

```
select col1,col2 into #t from t where 1=0
```

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：

```
create table #t(...)
```

13.很多时候用 exists 代替 in 是一个好的选择：

```
select num from a where num in(select num from b)
```

用下面的语句替换：

```
select num from a where exists(select 1 from b where num=a.num)
```

14.并不是所有索引对查询都有效，SQL 是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL 查询可能不会去利用索引，如一表中有字段 sex，male.female 几乎各一半，那么即使在 sex 上建了索引也对查询效率起不了作用。

15.索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过 6 个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

16.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

17.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够

了。

18.尽可能的使用 `varchar/nvarchar` 代替 `char/nchar` , 因为首先变长字段存储空间小,可以节省存储空间,其次对于查询来说,在一个相对较小的字段内搜索效率显然要高些。

19.任何地方都不要使用 `select * from t` ,用具体的字段列表代替 “*”,不要返回用不到的任何字段。

20.尽量使用表变量来代替临时表。如果表变量包含大量数据, 请注意索引非常有限 (只有主键索引)。

21.避免频繁创建和删除临时表, 以减少系统表资源的消耗。

22.临时表并不是不可使用,适当地使用它们可以使某些例程更有效,例如,当需要重复引用大型表或常用表中的某个数据集时。但是,对于一次性事件,较好使用导出表。

23.在新建临时表时,如果一次性插入数据量很大,那么可以使用 `select into` 代替 `create table` ,避免造成大量 `log` ,以提高速度;如果数据量不大,为了缓和系统表的资源,应先 `create table` ,然后 `insert`。

24.如果使用到了临时表,在存储过程的最后务必将所有的临时表显式删除,先 `truncate table` ,然后 `drop table` ,这样可以避免系统表的较长时间锁定。

25.尽量避免使用游标,因为游标的效率较差,如果游标操作的数据超过 1 万行,那么就应该考虑改写。

26.使用基于游标的方法或临时表方法之前,应先寻找基于集的解决方案来解决问题,基于集的方法通常更有效。

27.与临时表一样,游标并不是不可使用。对小型数据集使用

FAST_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

28.在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONEINPROC 消息。

29.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

30.尽量避免大事务操作，提高系统并发能力。

3.18 数据库优化方案整理

1.优化说明

(1)有数据表明，用户可以承受的最大等待时间为 8 秒。数据库优化策略有很多，设计初期，建立好的数据结构对于后期性能优化至关重要。因为数据库结构是系统的基石，基础打不好，使用各种优化策略，也不能达到很完美的效果。

(2)数据库优化的几个方面



可以看出来，数据结构.SQL.索引是成本最低，且效果最好的优化手段。

(3)性能优化是无止境的，当性能可以满足需求时即可，不要过度优化。

2.优化方向

(1)SQL 以及索引的优化

首先要根据需求写出结构良好的 SQL，然后根据 SQL 在表中建立有效的索引。但是如果索引太多，不但会影响写入的效率，对查询也有一定的影响。

(2)合理的数据库是设计

根据数据库三范式来进行表结构的设计。设计表结构时，就需要考虑如何设计才能更有效的查询。

数据库三范式：

第一范式：数据表中每个字段都必须是不可拆分的最小单元，也就是确保每一列的原子性；

第二范式：满足一范式后，表中每一列必须有唯一性，都必须依赖于主键；

第三范式：满足二范式后，表中的每一列只与主键直接相关而不是间接相关(外键也是直接相关)，字段没有冗余。

注意：没有最好的设计，只有最合适的设计，所以不要过分注重理论。三范式可以作为一个基本依据，不要生搬硬套。

有时候可以根据场景合理地反规范化：

A：分割表。

B：保留冗余字段。当两个或多个表在查询中经常需要连接时，可以在其中一个表上增加若干冗余的字段，以避免表之间的连接过于频繁，一般在冗余列的数据不经常变动的情况下使用。

C：增加派生列。派生列是由表中的其它多个列的计算所得，增加派生列可以减少统计运算，在数据汇总时可以大大缩短运算时间。

数据库五大约束：

A：PRIMARY key:设置主键约束；

B：UNIQUE：设置唯一性约束，不能有重复值；

C：DEFAULT 默认值约束

D：NOT NULL：设置非空约束，该字段不能为空；

E：FOREIGN key :设置外键约束。

字段类型选择：

A：尽量使用 TINYINT.SMALLINT.MEDIUM_INT 作为整数类型而非 INT，如果非负则

加上 UNSIGNED

B：VARCHAR 的长度只分配真正需要的空间

C：使用枚举或整数代替字符串类型

D：尽量使用 TIMESTAMP 而非 DATETIME

E：单表不要有太多字段，建议在 20 以内

F：避免使用 NULL 字段，很难查询优化且占用额外索引空间

(3)系统配置的优化

例如：MySQL 数据库 my.cnf

(4)硬件优化

更快的 IO.更多的内存。一般来说内存越大，对于数据库的操作越好。但是 CPU 多就不一定了，因为他并不会用到太多的 CPU 数量，有很多的查询都是单 CPU。另外使用高的 IO (SSD.RAID)，但是 IO 并不能减少数据库锁的机制。所以说如果查询缓慢是因为数据库内部的一些锁引起的，那么硬件优化就没有什么意义。

3. 优化方案

代码优化

之所以把代码放到第一位，是因为这一点最容易引起技术人员的忽视。很多技术人员拿到一个性能优化的需求以后，言必称缓存.异步.JVM 等。实际上，第一步就应该是分析相关的代码，找出相应的瓶颈，再来考虑具体的优化策略。有一些性能问题，完全是由于代码写的不合理，通过直接修改一下代码就能解决问题的，比如 for 循环次数过多.作了很多无谓的条件判断.相同逻辑重复多次等。

举个例子：

一个 update 操作，先查询出 entity，再执行 update，这样无疑多了一次数据库交互。还有一个问题，update 语句可能会操作一些无需更新的字段。

我们可以将表单中涉及到的属性，以及 updateTime，updateUser 等赋值

到 entity , 直接通过 pdateByPrimaryKeySelective , 去 update 特定字段

定位慢 SQL , 并优化

这是最常用.每一个技术人员都应该掌握基本的 SQL 调优手段 (包括方法.工具.辅助系统等)。这里以 MySQL 为例 , 最常见的方式是 , 由自带的慢查询日志或者开源的慢查询系统定位到具体的出问题的 SQL 然后使用 explain.profile 等工具来逐步调优 , 最后经过测试达到效果后上线。

SqlServer 执行计划 :

通过执行计划 , 我们能得到哪些信息 :

A : 哪些步骤花费的成本比较高

B : 哪些步骤产生的数据量多 , 数据量的多少用线条的粗细表示 , 很直观

C : 每一步执行了什么动作

具体优化手段 :

A : 尽量少用 (或者不用) sqlserver 自带的函数

select id from t where substring(name,1,3) = ' abc'

select id from t where datediff(day,createdate,' 2005-11-30') = 0

可以这样查询 :

select id from t where name like 'abc%'

select id from t where createdate >= '2005-11-30' and createdate
< '2005-12-1'

B : 连续数值条件 , 用 BETWEEN 不用 IN :SELECT id FROM t WHERE num
BETWEEN 1 AND 5

C : Update 语句 , 如果只更改 1.2 个字段 , 不要 Update 全部字段 , 否则

频繁调用会引起明显的性能消耗

D：尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型

E：不建议使用 `select * from t`，用具体的字段列表代替“*”，不要返回用不到的任何字段。尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理

F：表与表之间通过一个冗余字段来关联，要比直接使用 JOIN 有更好的性能

G：`select count(*) from table`；这样不带任何条件的 count 会引起全表扫描

连接池调优

我们的应用为了实现数据库连接的高效获取.对数据库连接的限流等目的，通常会采用连接池类的方案，即每一个应用节点都管理了一个到各个数据库的连接池。随着业务访问量或者数据量的增长，原有的连接池参数可能不能很好地满足需求，这个时候就需要结合当前使用连接池的原理.具体的连接池监控数据和当前的业务量作一个综合的判断，通过反复的几次调试得到最终的调优参数。

合理使用索引

索引一般情况下都是高效的。但是由于索引是以空间换时间的一种策略，索引本身在提高查询效率的同时会影响插入.更新.删除的效率，频繁写的表不宜建索引。

选择合适的索引列，选择在 `where`，`group by`，`order by`，`on` 从句中出现的列作为索引项，对于离散度不大的列没有必要创建索引。

主键已经是索引了，所以 primary key 的主键不用再设置 unique 唯一索引

索引类型

主键索引 (PRIMARY KEY)

唯一索引 (UNIQUE)

普通索引 (INDEX)

组合索引 (INDEX)

全文索引 (FULLTEXT)

可以应用索引的操作符

大于等于

Between

IN

LIKE 不以 % 开头

不能应用索引的操作符

NOT IN

LIKE %_ 开头

如何选择索引字段

A：字段出现在查询条件中，并且查询条件可以使用索引

B：通常对数字的索引和检索要比对字符串的索引和检索效率更高

C：语句执行频率高，一天会有几千次以上

D：通过字段条件可筛选的记录集很小

无效索引

A：尽量不要在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索

引而进行全表扫描

B：应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引

而进行全表扫描。

C：应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描

```
select id from t where num=10 or Name = 'admin'
```

可以这样查询：

```
select id from t where num = 10
```

```
union
```

```
select id from t where Name = 'admin'
```

union all 返回所有数据，不管是不是重复。 union 会自动压缩，去除重复数据。

D：不做列运算

where age + 1 = 10，任何对列的操作都将导致表扫描，它包括数据库教程函数.计算

表达式等

E：查询 like，如果是 '%aaa' 不会使用到索引

分表

分表方式

水平分割（按行）.垂直分割(按列)

分表场景

A： 根据经验，MySQL 表数据一般达到百万级别，查询效率就会很低。

B： 一张表的某些字段值比较大并且很少使用。可以将这些字段隔离成单独一张表，通过外键关联，例如考试成绩，我们通常关注分数，不关注考试详情。

水平分表策略

按时间分表：当数据有很强的实效性，例如微博的数据，可以按月分割。

按区间分表：例如用户表 1 到一百万用一张表，一百万到两百万用一张表。

hash 分表：通过一个原始目标 id 或者是名称按照一定的 hash 算法计算出数据存储的表名。

读写分离

当一台服务器不能满足需求时，采用读写分离【写: update/delete/add】的方式进行集群。

一台数据库支持最大连接数是有限的，如果用户的并发访问很多，一台服务器无法满足需求，可以集群处理。MySQL 集群处理技术最常用的就是读写分离。

主从同步：数据库最终会把数据持久化到磁盘，集群必须确保每个数据库服务器的数据是一致的。从库读主库写，从库从主库上同步数据。

读写分离：使用负载均衡实现，写操作都往主库上写，读操作往从服务器上读。

缓存

缓存分类

本地缓存：HashMap/ConcurrentHashMap.Ehcache.Guava Cache 等

缓存服务：Redis/Tair/Memcache 等

使用场景

短时间内相同数据重复查询多次且数据更新不频繁，这个时候可以选择先从缓存查询，查询不到再从数据库加载并回设到缓存的方式。此种场景较适合用单机缓存。

高并发查询热点数据，后端数据库不堪重负，可以用缓存来扛。

缓存作用：减轻数据库的压力，减少访问时间。

缓存选择：如果数据量小，并且不会频繁地增长又清空（这会导致频繁地垃圾回收），那么可以选择本地缓存。具体的话，如果需要一些策略的支持（比如缓存满的逐出策略），可以考虑 Ehcache；如不需要，可以考虑 HashMap；如需要考虑多线程并发的场景，可以考虑 ConcurrentHashMap。

其他情况，可以考虑缓存服务。目前从资源的投入度、可运维性、是否能动态扩容以及配套设施来考虑，我们优先考虑 Tair。除非目前 Tair 还不能支持的场合（比如分布式锁、Hash 类型的 value），我们考虑用 Redis。

缓存穿透一般的缓存系统，都是按照 key 去缓存查询，如果不存在对应的 value，就应该去后端系统查找（比如 DB）。如果 key 对应的 value 是一定不存在的，并且对该 key 并发请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

对查询结果为空的情况也进行缓存，缓存时间设置短点，或者该 key 对应的数据 insert 了之后清理缓存。

缓存并发有时候如果网站并发访问高，一个缓存如果失效，可能出现多个进程同时查询 DB，同时设置缓存的情况，

如果并发确实很大，这也可能造成 DB 压力过大，还有缓存频繁更新的问题。

对缓存查询加锁，如果 KEY 不存在，就加锁，然后查 DB 入缓存，然后解锁；其他进程如果有锁就等待，然后等解锁后返回数据或者进入 DB 查询。

缓存雪崩(失效)

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给

后端系统(比如 DB)

带来很大压力。

不同的 key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

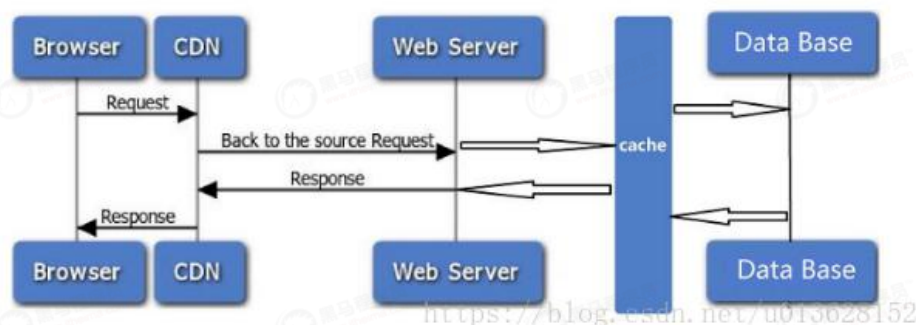
防止缓存空间不够用

(1) 给缓存服务，选择合适的缓存逐出算法，比如最常见的 LRU。

(2) 针对当前设置的容量，设置适当的警戒值，比如 10G 的缓存，当缓存数据达到 8G 的时候，就开始发出报警，提前排查问题或者扩容。

(3) 给一些没有必要长期保存的 key，尽量设置过期时间。

我们看下图，在 WebServer(Dao 层) 和 DB 之间加一层 cache，这层 cache 一般选取的介质是内存，因为我们都知道存入数据库的数据都具有持久化的特点，那么读写会有磁盘 IO 的操作，内存的读写速度远比磁盘快得多。(选用存储介质，提高访问速度：内存>>磁盘；减少磁盘 IO 的操作，减少重复查询，提高吞吐量)



常用开源的缓存工具有：ehcache.memcache.Redis。

ehcache 是一个纯 Java 的进程内缓存框架，hibernate 使用其做二级缓存。

同时，ehcache 可以通过多播的方式实现集群。本人主要用于本地的缓存，数据库上层的缓存。

memcache 是一套分布式的高速缓存系统，提供 key-value 这样简单的数

据储存，可充分利用 CPU 多核，无持久化功能。在做 web 集群中可以用做 session 共享，页面对象缓存。

Redis 高性能的 key-value 系统，提供丰富的数据类型，单核 CPU 有抗并发能力，有持久化和主从复制的功能。本人主要使用 Redis 的 Redis sentinel，根据不同业务分为多组。

Redis 注意事项

A：在增加 key 的时候尽量设置过期时间，不然 Redis Server 的内存使用会达到系统物理内存的最大值，导致 Redis 使用 VM 降低系统性能；

B：Redis Key 设计时应该尽可能短，Value 尽量不要使用复杂对象；

C：将对象转换成 JSON 对象（利用现成的 JSON 库）后存入 Redis；

D：将对象转换成 Google 开源二进制协议对象（Google Protobuf，和 JSON 数据格式类似，但是因为是二进制表现，所以性能效率以及空间占用都比 JSON 要小；缺点是 Protobuf 的学习曲线比 JSON 大得多）；

E：Redis 使用完以后一定要释放连接。

读取缓存中是否有相关数据，如果缓存中有相关数据，则直接返回，这就是所谓的数据命中 “hit”

如果缓存中没有相关数据，则从数据库读取相关数据，放入缓存中，再返回。这就是所谓的数据未命中 “miss”

缓存的命中率 = 命中缓存请求个数/总缓存访问请求个数 =
$$\text{hit}/(\text{hit}+\text{miss})$$

NoSQL

与缓存的区别

先说明一下,这里介绍的和缓存不一样,虽然 Redis 等也可以用来做数据存储方案(比如 Redis 或者 Tair),但 NoSql 是把它作为 DB 来用。如果当作 DB 来用,需要有效保证数据存储方案的可用性.可靠性。

使用场景

需要结合具体的业务场景,看这块业务涉及的数据是否适合用 NoSQL 来存储,对数据的操作方式是否适合用 NoSQL 的方式来操作,或者是否需要用到 NoSQL 的一些额外特性(比如原子加减等)。

如果业务数据不需要和其他数据作关联,不需要事务或者外键之类的支持,而且有可能写入会异常频繁,这个时候就比较适合用 NoSQL(比如 HBase)

比如,美团点评内部有一个对 exception 做的监控系统,如果在应用系统发生严重故障的时候,可能会短时间产生大量 exception 数据,这个时候如果选用 MySQL,会造成 MySQL 的瞬间写压力飙升,容易导致 MySQL 服务器的性能急剧恶化以及主从同步延迟之类的问题,这种场景就比较适合用 Hbase 类似的 NoSQL 来存储。

视图/存储过程

普通业务逻辑尽量不要使用存储过程,定时任务或报表统计函数可以根据团队资源情况采用存储过程处理。

GVM 调优

通过监控系统(如没有现成的系统,自己做一个简单的上报监控的系统也很容易)上对一些机器关键指标(gc time.gc count.各个分代的内存大小变化.机器的 Load 值与 CPU 使用率.JVM 的线程数等)的监控报警,也可以看 gc log 和 jstat 等命令的输出,再结合线上 JVM 进程服务的一些关键接口的性能数据和

请求体验，基本上就能定位出当前的 JVM 是否有问题，以及是否需要调优。

异步/多线程

针对某些客户端的请求，在服务端可能需要针对这些请求做一些附属的事情，这些事情其实用户并不关心或者用户不需要立即拿到这些事情的处理结果，这种情况就比较适合用异步的方式处理这些事情。

异步作用

A：缩短接口响应时间，使用户的请求快速返回，用户体验更好。

B：避免线程长时间处于运行状态，这样会引起服务线程池的可用线程长时间不够用，进而引起线程池任务队列长度增大，从而阻塞更多请求任务，使得更多请求得不到及时处理。

C：线程长时间处于运行状态，可能还会引起系统 Load、CPU 使用率、机器整体性能下降等一系列问题，甚至引发雪崩。异步的思路可以在不增加机器数和 CPU 数的情况下，有效解决这个问题。

异步实现

A：额外开辟线程，这里可以采用额外开辟一个线程或者使用线程池的做法，在 IO 线程（处理请求响应）之外的线程来处理相应的任务，在 IO 线程中让 response 先返回。

B：使用消息队列（MQ）中间件服务

搜索引擎

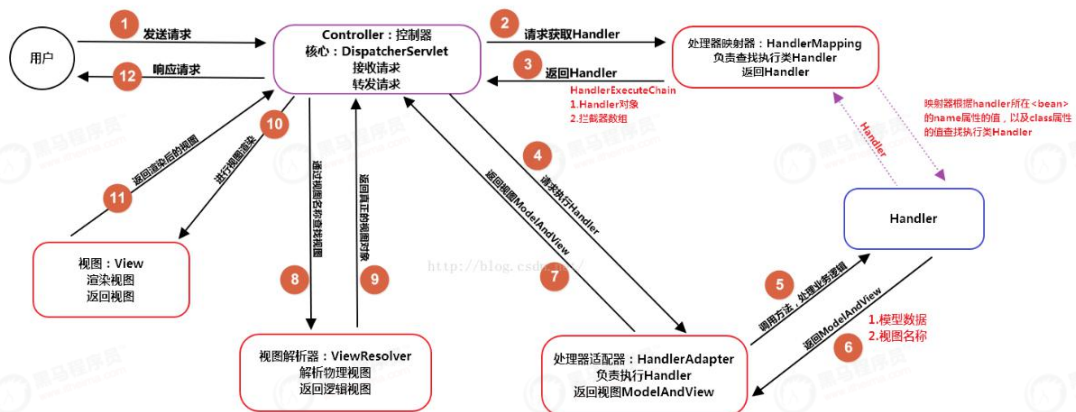
例如：solr，elasticsearch

四. SpringMVC 框架

4.1 什么是 SpringMVC ? 简单介绍下你对 SpringMVC 的理解?

SpringMVC 是一个基于 Java 的实现了 MVC 设计模式的请求驱动类型的轻量级 Web 框架, 通过把 Model, View, Controller 分离, 将 web 层进行职责解耦, 把复杂的 web 应用分成逻辑清晰的几部分, 简化开发, 减少出错, 方便组内开发人员之间的配合。

4.2 SpringMVC 的流程 ?



<https://blog.csdn.net/a745233700>

- (1) 用户发送请求至前端控制器 DispatcherServlet ;
- (2) DispatcherServlet 收到请求后, 调用 HandlerMapping 处理器映射器, 请求获取 Handle ;
- (3) 处理器映射器根据请求 url 找到具体的处理器, 生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet ;
- (4) DispatcherServlet 调用 HandlerAdapter 处理器适配器 ;
- (5) HandlerAdapter 经过适配调用 具体处理器(Handler, 也叫后端控

制器) ;

(6) Handler 执行完成返回 ModelAndView ;

(7) HandlerAdapter 将 Handler 执行结果 ModelAndView 返回给 DispatcherServlet ;

(8) DispatcherServlet 将 ModelAndView 传给 ViewResolver 视图解析器进行解析 ;

(9) ViewResolver 解析后返回具体 View ;

(10) DispatcherServlet 对 View 进行渲染视图 (即将模型数据填充至视图中)

(11) DispatcherServlet 响应用户。

4.3 SpringMVC 的优点

(1) 可以支持各种视图技术,而不仅仅局限于 JSP ;

(2) 与 Spring 框架集成 (如 IoC 容器.AOP 等);

(3) 清晰的角色分配 : 前端控制器(dispatcherServlet), 请求到处理器映射 (handlerMapping), 处理器适配器 (HandlerAdapter), 视图解析器 (ViewResolver)

(4) 支持各种请求资源的映射策略。

4.4 SpringMVC 的主要组件 ?

(1) 前端控制器 DispatcherServlet (不需要程序员开发)

作用 : 接收请求.响应结果 , 相当于转发器 , 有了 DispatcherServlet 就减少了其它组件之间的耦合度。

(2) 处理器映射器 HandlerMapping (不需要程序员开发)

作用：根据请求的 URL 来查找 Handler

(3) 处理器适配器 HandlerAdapter

注意：在编写 Handler 的时候要按照 HandlerAdapter 要求的规则去编写，这样适配器 HandlerAdapter 才可以正确的去执行 Handler。

(4) 处理器 Handler (需要程序员开发)

(5) 视图解析器 ViewResolver (不需要程序员开发)

作用：进行视图的解析，根据视图逻辑名解析成真正的视图 (view)

(6) 视图 View (需要程序员开发 jsp)

View 是一个接口，它的实现类支持不同的视图类型 (jsp , freemarker , pdf 等等)

4.5 SpringMVC 和 Struts2 的区别有哪些？

(1) SpringMVC 的入口是一个 servlet 即前端控制器 (DispatchServlet) , 而 struts2 入口是一个 filter 过滤器 (StrutsPrepareAndExecuteFilter) 。

(2) SpringMVC 是基于方法开发(一个 url 对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2 是基于类开发，传递参数是通过类的属性，只能设计为多例。

(3) Struts 采用值栈存储请求和响应的数据，通过 OGNL 存取数据，SpringMVC 通过参数解析器是将 request 请求内容解析，并给方法形参赋值，将数据和视图封装成 ModelAndView 对象，最后又将 ModelAndView 中的模型数据通过 request 域传输到页面。Jsp 视图解析器默认使用 jstl。

4.6 SpringMVC 怎么样设定重定向和转发的？

(1) 转发：在返回值前面加 "forward:"，譬如

"forward:user.do?name=method4"

(2) 重定向：在返回值前面加"redirect:"，譬如

"redirect:http://www.baidu.com"

4.7 SpringMVC 怎么和 Ajax 相互调用的？

通过 Jackson 框架就可以把 Java 里面的对象直接转化成 Js 可以识别的 Json

对象。具体步骤如下：

(1) 加入 Jackson.jar

(2) 在配置文件中配置 json 的映射

(3) 在接受 Ajax 方法里面可以直接返回 Object,List 等,但方法前面要加上

@ResponseBody 注解。

4.8 如何解决 Post 请求中文乱码问题，Get 的又如何处理呢？

(1) 解决 post 请求乱码问题：

在 web.xml 中配置一个 CharacterEncodingFilter 过滤器，设置成 utf-8；

```
<filter>
<filter-name>CharacterEncodingFilter</filter-name>
<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

    <init-param>
<param-name>encoding</param-name>
<param-value>utf-8</param-value>
</init-param>

</filter>

<filter-mapping>
<filter-name>CharacterEncodingFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

(2) get 请求中文参数出现乱码解决方法有两个：

修改 tomcat 配置文件添加编码与工程编码一致，如下：

```
<ConnectorURIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>
```

另外一种方法对参数进行重新编码：

```
String userName= new String(request.getParamter("userName").getBytes("ISO8859-1"),"utf-8")
```

ISO8859-1 是 tomcat 默认编码，需要将 tomcat 编码后的内容按 utf-8 编码。

4.9 SpringMVC 的异常处理 ？

答：可以将异常抛给 Spring 框架，由 Spring 框架来处理；我们只需要配置简单的异常处理器，在异常处理器中添视图页面即可。

4.10 SpringMVC 的控制器是不是单例模式,如果是,有什么问题,怎么解决？

答：是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

4.11 SpringMVC 常用的注解有哪些？

@RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。

@RequestBody：注解实现接收 http 请求的 json 数据，将 json 转换为 java 对象。

@ResponseBody 注解实现将 controller 方法返回对象转化为 json 对象响应给客户。

4.12 SpringMVC 中的控制器的注解一般用那个,有没有别的注解可以替代?

一般用@Controller 注解,表示是表现层,不能用别的注解代替。

4.13 如果在拦截请求中,我想拦截 get 方式提交的方法,怎么配置?

可以在@RequestMapping 注解里面加 method=RequestMethod.GET。

4.14 怎样在方法里面得到 Request,或者 Session?

直接在方法的形参中声明 request, SpringMVC 就自动把 request 对象传入。

4.15 如果想在拦截的方法里面得到从前台传入的参数,怎么得到?

直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样。

4.16 如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象?

直接在方法中声明这个对象, SpringMVC 就会自动把属性赋值到这个对象里面。

4.17 SpringMVC 中函数的返回值是什么?

返回值可以有很多类型,有 String, ModelAndView。ModelAndView 类把

视图和数据都合并在一起的，但一般用 String 比较好。

4.18 SpringMVC 用什么对象从后台向前台传递数据的？

通过 ModelMap 对象,可以在这个对象里面调用 put 方法,把对象加到里面,前台就可以通过 el 表达式拿到。

4.19 怎么样把 ModelMap 里面的数据放入 Session 里面？

可以在类上面加上@SessionAttributes 注解,里面包含的字符串就是要放入 session 里面的 key。

4.20 SpringMVC 里面拦截器是怎么写的？

有两种写法,一种是实现 HandlerInterceptor 接口,另外一种继承适配类,接着在接口方法当中,实现处理逻辑;然后在 SpringMVC 的配置文件中配置拦截器即可：

```
<!-- 配置 SpringMVC 的拦截器 -->
```

```
<mvc:interceptors>
```

```
<!-- 配置一个拦截器的 Bean 就可以了 默认是对所有请求都拦截 -->
```

```
<bean id="myInterceptor"  
class="com.zwp.action.MyHandlerInterceptor"></bean>
```

```
<!-- 只针对部分请求拦截 -->
```

```
<mvc:interceptor>
```

```
<mvc:mapping path="/modelMap.do" />
```

```
<bean class="com.zwp.action.MyHandlerInterceptorAdapter" />

</mvc:interceptor>

</mvc:interceptors>
```

4.21 注解原理

注解本质是一个继承了 Annotation 的特殊接口，其具体实现类是 Java 运行时生成的动态代理类。我们通过反射获取注解时，返回的是 Java 运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用 AnnotationInvocationHandler 的 invoke 方法。该方法会从 memberValues 这个 Map 中索引出对应的值。而 memberValues 的来源是 Java 常量池。

五. Spring 框架

5.1 Spring 是什么？

Spring 是一个轻量级的 IoC 和 AOP 容器框架。是为 Java 应用程序提供基础性服务的一套框架，目的是用于简化企业应用程序的开发，它使得开发者只需要关心业务需求。常见的配置方式有三种：基于 XML 的配置、基于注解的配置、基于 Java 的配置。

主要由以下几个模块组成：

Spring Core：核心类库，提供 IOC 服务；

Spring Context：提供框架式的 Bean 访问方式，以及企业级功能（JNDI、定时任务等）；

Spring AOP：AOP 服务；

Spring DAO：对 JDBC 的抽象，简化了数据访问异常的处理；

Spring ORM：对现有的 ORM 框架的支持；

Spring Web：提供了基本的面向 Web 的综合特性，例如多方文件上传；

Spring MVC：提供面向 Web 应用的 Model-View-Controller 实现。

5.2 Spring 的优点？

(1) Spring 属于低侵入式设计，代码的污染极低；

(2) Spring 的 DI 机制将对象之间的依赖关系交由框架处理，减低组件的耦合性；

(3) Spring 提供了 AOP 技术，支持将一些通用任务，如安全.事务.日志.权限等进行集中式管理，从而提供更好的复用。

(4) Spring 对于主流的应用框架提供了集成支持。

5.3 Spring 的 AOP 理解？

OOP 面向对象，允许开发者定义纵向的关系，但并不适用于定义横向的关系，导致了大量代码的重复，而不利于各个模块的重用。

AOP，一般称为面向切面，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证.日志.事务处理。

AOP 实现的关键在于 代理模式，AOP 代理主要分为静态代理和动态代理。静态代理的代表为 AspectJ；动态代理则以 Spring AOP 为代表。

(1) AspectJ 是静态代理的增强，所谓静态代理，就是 AOP 框架会在编译阶段生成 AOP 代理类，因此也称为编译时增强，他会在编译阶段将 AspectJ(切面)织入到 Java 字节码中，运行的时候就是增强之后的 AOP 对象。

(2) Spring AOP 使用的动态代理，所谓的动态代理就是说 AOP 框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个 AOP 对象，这个 AOP 对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

Spring AOP 中的动态代理主要有两种方式，JDK 动态代理和 CGLIB 动态代理：

(1)JDK 动态代理只提供接口的代理，不支持类的代理。核心 InvocationHandler 接口和 Proxy 类，InvocationHandler 通过 invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy 利用 InvocationHandler 动态创建一个符合某一接口的实例，生成目标类的代理对象。

(2)如果代理类没有实现 InvocationHandler 接口，那么 Spring AOP 会选择使用 CGLIB 来动态代理目标类。CGLIB (Code Generation Library)，是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现 AOP。CGLIB 是通过继承的方式做的动态代理，因此如果某个类被标记为 final，那么它是无法使用 CGLIB 做动态代理的。

(3) 静态代理与动态代理区别在于生成 AOP 代理对象的时机不同,相对来说 AspectJ 的静态代理方式具有更好的性能,但是 AspectJ 需要特定的编译器进行处理,而 Spring AOP 则无需特定的编译器处理。

InvocationHandler 的

invoke(Object proxy,Method method,Object[] args) :proxy 是最终生成的代理实例; method 是被代理目标实例的某个具体方法,args 是被代理目标实例某个方法的具体入参,在方法反射调用时使用。

5.4 Spring 的 IOC 理解 ?

(1) IOC 就是控制反转,是指创建对象的控制权的转移,以前创建对象的主动权和时机是由自己把控的,而现在这种权力转移到 Spring 容器中,并由容器根据配置文件去创建实例和管理各个实例之间的依赖关系,对象与对象之间松散耦合,也利于功能的复用。DI 依赖注入,和控制反转是同一个概念的不同角度的描述,即 应用程序在运行时依赖 IoC 容器来动态注入对象需要的外部资源。

(2) 最直观的表达就是,IOC 让对象的创建不用去 new 了,可以由 spring 自动生产,使用 java 的反射机制,根据配置文件在运行时动态的去创建对象以及管理对象,并调用对象的方法的。

(3) Spring 的 IOC 有三种注入方式 : 构造器注入.setter 方法注入.根据注解注入。

IOC 让相互协作的组件保持松散的耦合,而 AOP 编程允许你把遍布于应用各层的功能分离出来形成可重用的功能组件。

5.5 BeanFactory 和 ApplicationContext 有什么区别？

BeanFactory 和 ApplicationContext 是 Spring 的两大核心接口，都可以当做 Spring 的容器。其中 ApplicationContext 是 BeanFactory 的子接口。

(1) BeanFactory : 是 Spring 里面最底层的接口，包含了各种 Bean 的定义，读取 bean 配置文档，管理 bean 的加载、实例化，控制 bean 的生命周期，维护 bean 之间的依赖关系。ApplicationContext 接口作为 BeanFactory 的派生，除了提供 BeanFactory 所具有的功能外，还提供了更完整的框架功能：

继承 MessageSource，因此支持国际化。

统一的资源文件访问方式。

提供在监听器中注册 bean 的事件。

同时加载多个配置文件。

载入多个（有继承关系）上下文，使得每一个上下文都专注于一个特定的层次，比如应用的 web 层。

(2) BeanFactory 采用的是延迟加载形式来注入 Bean 的，即只有在使用到某个 Bean 时(调用 getBean())，才对该 Bean 进行加载实例化。这样，我们就不能发现一些存在的 Spring 的配置问题。如果 Bean 的某一个属性没有注入，BeanFactory 加载后，直至第一次使用调用 getBean 方法才会抛出异常。

ApplicationContext，它是在容器启动时，一次性创建了所有的 Bean。这样，在容器启动时，我们就可以发现 Spring 中存在的配置错误，这样有利于检查所依赖属性是否注入。ApplicationContext 启动后预载入所有的单实例 Bean，通过预载入单实例 bean，确保当你需要的时候，你就不用等待，因为它们已经创建好了。

相对于基本的 BeanFactory , ApplicationContext 唯一的不足是占用内存空间。当应用程序配置 Bean 较多时 , 程序启动较慢。

(3) BeanFactory 通常以编程的方式被创建 , ApplicationContext 还能以声明的方式创建 , 如使用 ContextLoader。

(4) BeanFactory 和 ApplicationContext 都支持 BeanPostProcessor.BeanFactoryPostProcessor 的使用 , 但两者之间的区别是 : BeanFactory 需要手动注册 , 而 ApplicationContext 则是自动注册。

5.6 请解释 Spring Bean 的生命周期 ?

首先说一下 Servlet 的生命周期 : 实例化 , 初始 init , 接收请求 service , 销毁 destroy ;

Spring 上下文中的 Bean 生命周期也类似 , 如下 :

(1) 实例化 Bean :

对于 BeanFactory 容器 , 当客户向容器请求一个尚未初始化的 bean 时 , 或初始化 bean 的时候需要注入另一个尚未初始化的依赖时 , 容器就会调用 createBean 进行实例化。对于 ApplicationContext 容器 , 当容器启动结束后 , 通过获取 BeanDefinition 对象中的信息 , 实例化所有的 bean。

(2) 设置对象属性 (依赖注入) :

实例化后的对象被封装在 BeanWrapper 对象中 , 紧接着 , Spring 根据 BeanDefinition 中的信息 以及 通过 BeanWrapper 提供的设置属性的接口完成依赖注入。

(3) 处理 Aware 接口 :

接着，Spring 会检测该对象是否实现了 xxxAware 接口，并将相关的 xxxAware 实例注入给 Bean：

如果这个 Bean 已经实现了 BeanNameAware 接口，会调用它实现的 setBeanName(String beanId)方法，此处传递的就是 Spring 配置文件中 Bean 的 id 值；

如果这个 Bean 已经实现了 BeanFactoryAware 接口，会调用它实现的 setBeanFactory()方法，传递的是 Spring 工厂自身。

如果这个 Bean 已经实现了 ApplicationContextAware 接口，会调用 setApplicationContext(ApplicationContext)方法，传入 Spring 上下文；

(4) BeanPostProcessor：

如果想对 Bean 进行一些自定义的处理，那么可以让 Bean 实现了 BeanPostProcessor 接口，那将会调用 postProcessBeforeInitialization(Object obj, String s)方法。由于这个方法是在 Bean 初始化结束时调用的，所以可以被应用于内存或缓存技术；

(5) InitializingBean 与 init-method：

如果 Bean 在 Spring 配置文件中配置了 init-method 属性，则会自动调用其配置的初始化方法。

(6) 如果这个 Bean 实现了 BeanPostProcessor 接口，将会调用 postProcessAfterInitialization(Object obj, String s)方法；

以上几个步骤完成后，Bean 就已经被正确创建了，之后就可以使用这个 Bean 了。

(7) DisposableBean：

当 Bean 不再需要时,会经过清理阶段 如果 Bean 实现了 DisposableBean 这个接口,会调用其实现的 destroy()方法;

(8) destroy-method :

最后,如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性,会自动调用其配置的销毁方法。

5.7 解释 Spring 支持的几种 bean 的作用域。

Spring 容器中的 bean 可以分为 5 个范围 :

(1) singleton : 默认,每个容器中只有一个 bean 的实例,单例的模式由 BeanFactory 自身来维护。

(2) prototype : 为每一个 bean 请求提供一个实例。

(3) request : 为每一个网络请求创建一个实例,在请求完成以后,bean 会失效并被垃圾回收器回收。

(4) session : 与 request 范围类似,确保每个 session 中有一个 bean 的实例,在 session 过期后,bean 会随之失效。

(5) global-session : 全局作用域,global-session 和 Portlet 应用相关。

当你的应用部署在 Portlet 容器中工作时,它包含很多 portlet。如果你想要声明让所有的 portlet 共用全局的存储变量的话,那么这全局变量需要存储在 global-session 中。全局作用域与 Servlet 中的 session 作用域效果相同。

5.8 使用注解之前要开启自动扫描功能

其中 base-package 为需要扫描的包(含子包)。

```
<context:component-scan base-package="cn.test"/>
```

@Configuration 把一个类作为一个 IoC 容器，它的某个方法头上如果注册了@Bean，就会作为这个 Spring 容器中的 Bean。

@Scope 注解 作用域

@Lazy(true) 表示延迟初始化

@Service 用于标注业务层组件。

@Controller 用于标注控制层组件（如 struts 中的 action）

@Repository 用于标注数据访问组件，即 DAO 组件。

@Component 泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

@Scope 用于指定 scope 作用域的（用在类上）

@PostConstruct 用于指定初始化方法（用在方法上）

@PreDestroy 用于指定销毁方法（用在方法上）

@Resource 默认按名称装配，当找不到与名称匹配的 bean 才会按类型装配。

@DependsOn：定义 Bean 初始化及销毁时的顺序

@Primary：自动装配时当出现多个 Bean 候选者时，被注解为@Primary 的 Bean 将作为首选者，否则将抛出异常

@Autowired 默认按类型装配，如果我们想使用按名称装配，可以结合@Qualifier 注解一起使用。如下：

@Autowired @Qualifier("personDaoBean") 存在多个实例配合使用

5.9 Spring 框架中的单例 Beans 是线程安全的么？

Spring 框架并没有对单例 bean 进行任何多线程的封装处理。关于单例 bean 的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的 Spring bean 并没有可变的状态(比如 Servlet 类和 DAO 类)，所以在某种程度上说 Spring 的单例 bean 是线程安全的。如果你的 bean 有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。最浅显的解决办法就是将多态 bean 的作用域由“singleton”变更为“prototype”。

5.10 Spring 如何处理线程并发问题？

在一般情况下，只有无状态的 Bean 才可以在多线程环境下共享，在 Spring 中，绝大部分 Bean 都可以声明为 singleton 作用域，因为 Spring 对一些 Bean 中非线程安全状态采用 ThreadLocal 进行处理，解决线程安全问题。

ThreadLocal 和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而 ThreadLocal 采用了“空间换时间”的方式。

ThreadLocal 会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。

ThreadLocal 提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进 ThreadLocal。

5.11 Spring 基于 xml 注入 bean 的几种方式

(1) Set 方法注入；

(2) 构造器注入：1.通过 index 设置参数的位置；2.通过 type 设置参数类型；3.通过 name 注入

(3) 静态工厂注入；

(4) 实例工厂；

5.12 Spring 的自动装配：

在 Spring 中，对象无需自己查找或创建与其关联的其他对象，由容器负责把需要相互协作的对象引用赋予各个对象，使用 [autowire](#) 来配置自动装载模式。

在 Spring 框架 xml 配置中共有 5 种自动装配：

(1) no：默认的方式是不进行自动装配的，通过手工设置 ref 属性来进行装配 bean。

(2) byName：通过 bean 的名称进行自动装配，如果一个 bean 的 property 与另一 bean 的 name 相同，就进行自动装配。

(3) byType：通过参数的数据类型进行自动装配。

(4) constructor：利用构造函数进行装配，并且构造函数的参数通过 byType 进行装配。

(5) autodetect：自动探测，如果有构造方法，通过 construct 的方式自动装配，否则使用 byType 的方式自动装配。

基于注解的方式：

使用 @Autowired 注解来自动装配指定的 bean。在使用 @Autowired 注解之前需要在 Spring 配置文件进行配置，`<context:annotation-config />`。在启动 spring IoC 时，容器自动装载了一个 AutowiredAnnotationBeanPostProcessor 后置处理器，当容器扫描到 @Autowired、@Resource 或 @Inject 时，就会在 IoC 容器自动查找需要的 bean，并装配给该对象的属性。在使用 @Autowired 时，首先在容器中查询对应类型的 bean：

如果查询结果刚好为一个，就将该 bean 装配给 @Autowired 指定的数据；

如果查询的结果不止一个，那么 @Autowired 会根据名称来查找；

如果上述查找的结果为空，那么会抛出异常。解决方法时，使用 `required=false`。

`@Autowired` 可用于：构造函数.成员变量.Setter 方法

注：`@Autowired` 和 `@Resource` 之间的区别

(1) `@Autowired` 默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它 `required` 属性为 `false`）。

(2) `@Resource` 默认是按照名称来装配注入的，只有当找不到与名称匹配的 bean 才会按照类型来装配注入。

5.13 Spring 框架中都用了哪些设计模式？

(1) 工厂模式：`BeanFactory` 就是简单工厂模式的体现，用来创建对象的实例；

(2) 单例模式：`Bean` 默认为单例模式。

(3) 代理模式：`Spring` 的 AOP 功能用到了 `JDK` 的动态代理和 `CGLIB` 字节码生成技术；

(4) 模板方法：用来解决代码重复的问题。比如 `RestTemplate`, `JmsTemplate`, `JpaTemplate`。

(5) 观察者模式：定义对象间一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如 `Spring` 中 `listener` 的实现 --`ApplicationListener`。

5.14 Spring 事务的实现方式和实现原理

Spring 事务的本质其实就是数据库对事务的支持,没有数据库的事务支持, spring 是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过 binlog 或者 redo log 实现的。

(1) Spring 事务的种类 :

spring 支持编程式事务管理和声明式事务管理两种方式 :

A.编程式事务管理使用 TransactionTemplate。

B.声明式事务管理建立在 AOP 之上的。其本质是通过 AOP 功能,对方法前后进行拦截,将事务处理的功能编织到拦截的方法中,也就是在目标方法开始之前加入一个事务,在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就是不需要在业务逻辑代码中掺杂事务管理的代码,只需在配置文件中做相关的事务规则声明或通过@Transactional 注解的方式,便可以将事务规则应用到业务逻辑中。

声明式事务管理要优于编程式事务管理,这正是 spring 倡导的非侵入式的开发方式,使业务代码不受污染,只要加上注解就可以获得完全的事务支持。唯一不足地方是,最细粒度只能作用到方法级别,无法做到像编程式事务那样可以作用到代码块级别。

(2) Spring 的事务传播行为 :

Spring 事务的传播行为说的是,当多个事务同时存在的时候, Spring 如何处理这些事务的行为。

① PROPAGATION_REQUIRED :如果当前没有事务,就创建一个新事务,如果当前存在事务,就加入该事务,该设置是最常用的设置。

② PROPAGATION_SUPPORTS : 支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。

③ PROPAGATION_MANDATORY : 支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ PROPAGATION_REQUIRES_NEW : 创建新事务，无论当前存不存在事务，都创建新事务。

⑤ PROPAGATION_NOT_SUPPORTED : 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

⑥ PROPAGATION_NEVER : 以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ PROPAGATION_NESTED : 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按 REQUIRED 属性执行。

(3) Spring 中的隔离级别 :

① ISOLATION_DEFAULT : 这是个 PlatformTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。

② ISOLATION_READ_UNCOMMITTED : 读未提交，允许另外一个事务可以看到这个事务未提交的数据。

③ ISOLATION_READ_COMMITTED : 读已提交，保证一个事务修改的数据提交后才能被另一事务读取，而且能看到该事务对已有记录的更新。

④ ISOLATION_REPEATABLE_READ : 可重复读，保证一个事务修改的数据提交后才能被另一事务读取，但是不能看到该事务对已有记录的更新。

⑤ ISOLATION_SERIALIZABLE : 一个事务在执行的过程中完全看不到其他事务对数据库所做的更新。

5.15 Spring 框架中有哪些不同类型的事件？

Spring 提供了以下 5 种标准的事件：

(1) 上下文更新事件 (ContextRefreshedEvent) : 在调用 ConfigurableApplicationContext 接口中的 refresh()方法时被触发。

(2) 上下文开始事件 (ContextStartedEvent) : 当容器调用 ConfigurableApplicationContext 的 Start()方法开始/重新开始容器时触发该事件。

(3) 上下文停止事件 (ContextStoppedEvent) : 当容器调用 ConfigurableApplicationContext 的 Stop()方法停止容器时触发该事件。

(4) 上下文关闭事件 (ContextClosedEvent) : 当 ApplicationContext 被关闭时触发该事件。容器被关闭时，其管理的所有单例 Bean 都被销毁。

(5) 请求处理事件 (RequestHandledEvent) : 在 Web 应用中，当一个 http 请求 (request) 结束触发该事件。

如果一个 bean 实现了 ApplicationListener 接口，当一个 ApplicationEvent 被发布以后，bean 会自动被通知。

5.16 解释一下 Spring AOP 里面的几个名词

(1)切面(Aspect) :被抽取的公共模块 ,可能会横切多个对象。在 Spring AOP 中 ,切面可以使用通用类 (基于模式的风格) 或者在普通类中以 @AspectJ 注解来实现。

(2)连接点 (Join point) :指方法 ,在 Spring AOP 中 ,一个连接点 总是 代表一个方法的执行。

(3)通知 (Advice) :在切面的某个特定的连接点 (Join point) 上执行的动作。通知有各种类型 ,其中包括 “around” . “before” 和 “after” 等通知。许多 AOP 框架 ,包括 Spring ,都是以拦截器做通知模型 , 并维护一个以连接点为中心的拦截器链。

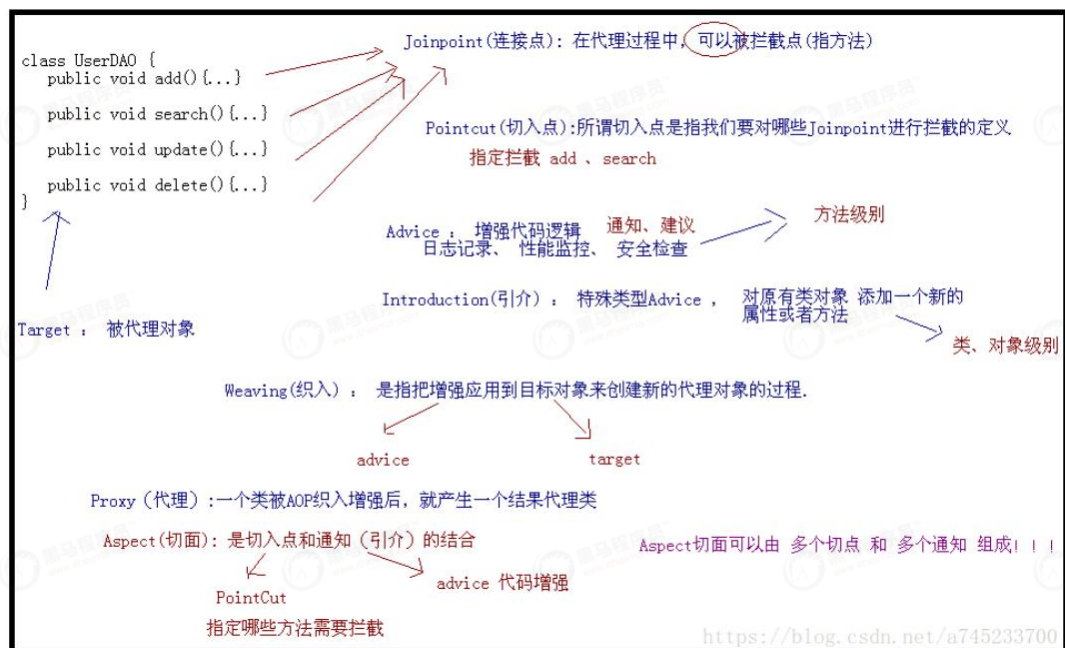
(4)切入点 (Pointcut) :切入点是指 我们要对哪些 Join point 进行拦截的定义。通过切入点表达式 ,指定拦截的方法 ,比如指定拦截 add*.search*。

(5)引入 (Introduction) : (也被称为内部类型声明 (inter-type declaration))。声明额外的方法或者某个类型的字段。Spring 允许引入新的接口 (以及一个对应的实现) 到任何被代理的对象。例如 ,你可以使用一个引入来使 bean 实现 IsModified 接口 ,以便简化缓存机制。

(6)目标对象 (Target Object) : 被一个或者多个切面 (aspect) 所通知 (advise) 的对象。也有人把它叫做 被通知 (adviced) 对象。既然 Spring AOP 是通过运行时代理实现的 ,这个对象永远是一个 被代理 (proxied) 对象。

(7)织入 (Weaving) :指把增强应用到目标对象来创建新的代理对象的过程。Spring 是在运行时完成织入。

切入点 (pointcut) 和连接点 (join point) 匹配的概念是 AOP 的关键，这使得 AOP 不同于其它仅提供拦截功能的旧技术。切入点使得定位通知 (advice) 可独立于 OO 层次。例如，一个提供声明式事务管理的 around 通知可以被应用到一组横跨多个对象中的方法上 (例如服务层的所有业务操作)。



5.17 Spring 通知有哪些类型？

(1) 前置通知 (Before advice)：在某连接点 (join point) 之前执行的通知，但这个通知不能阻止连接点前的执行 (除非它抛出一个异常)。

(2) 返回后通知 (After returning advice)：在某连接点 (join point) 正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。

(3) 抛出异常后通知 (After throwing advice)：在方法抛出异常退出时执行的通知。

(4) 后通知 (After (finally) advice)：当某连接点退出的时候执行的通知 (不论是正常返回还是异常退出)。

(5) 环绕通知 (Around Advice)：包围一个连接点 (join point) 的通知，如方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它

也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。环绕

通知是最常用的一种通知类型。大部分基于拦截的 AOP 框架，例如 Nanning 和 JBoss4，都只提供环绕通知。

同一个 aspect，不同 advice 的执行顺序：

①没有异常情况下的执行顺序：

around before advice

before advice

target method 执行

around after advice

after advice

afterReturning

②有异常情况下的执行顺序：

around before advice

before advice

target method 执行

around after advice

after advice

afterThrowing:异常发生

java.lang.RuntimeException: 异常发生

六. Mybatis 框架

6.1 什么是 Mybatis ?

(1)Mybatis 是一个半 ORM(对象关系映射) 框架 , 它内部封装了 JDBC , 开发时只需要关注 SQL 语句本身 , 不需要花费精力去处理加载驱动.创建连接.创建 statement 等繁杂的过程。程序员直接编写原生态 sql , 可以严格控制 sql 执行性能 , 灵活度高。

(2)MyBatis 可以使用 XML 或注解来配置和映射原生信息 , 将 POJO 映射成数据库中的记录 , 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。

(3) 通过 xml 文件或注解的方式将要执行的各种 statement 配置起来 , 并通过 java 对象和 statement 中 sql 的动态参数进行映射生成最终执行的 sql 语句 , 最后由 mybatis 框架执行 sql 并将结果映射为 java 对象并返回。(从执行 sql 到返回 result 的过程) 。

6.2 Mybaits 的优点

(1) 基于 SQL 语句编程 , 相当灵活 , 不会对应用程序或者数据库的现有设计造成任何影响 , SQL 写在 XML 里 , 解除 sql 与程序代码的耦合 , 便于统一管理 ; 提供 XML 标签 , 支持编写动态 SQL 语句 , 并可重用。

(2) 与 JDBC 相比 , 减少了 50% 以上的代码量 , 消除了 JDBC 大量冗余的代码 , 不需要手动开关连接 ;

(3) 很好的与各种数据库兼容 (因为 MyBatis 使用 JDBC 来连接数据库 , 所以只要 JDBC 支持的数据库 MyBatis 都支持) 。

(4) 能够与 Spring 很好的集成 ;

(5) 提供映射标签 , 支持对象与数据库的 ORM 字段关系映射 ; 提供对象关系映射标签 , 支持对象关系组件维护。

6.3 MyBatis 框架的缺点

(1) SQL 语句的编写工作量较大 , 尤其当字段多.关联表多时 , 对开发人员编写 SQL 语句的功底有一定要求。

(2) SQL 语句依赖于数据库 , 导致数据库移植性差 , 不能随意更换数据库。

6.4 MyBatis 框架适用场合

(1) MyBatis 专注于 SQL 本身 , 是一个足够灵活的 DAO 层解决方案。

(2) 对性能的要求很高 , 或者需求变化较多的项目 , 如互联网项目 , MyBatis 将是不错的选择。

6.5 MyBatis 与 Hibernate 有哪些不同 ?

(1) Mybatis 和 hibernate 不同 , 它不完全是个 ORM 框架 , 因为 MyBatis 需要程序员自己编写 Sql 语句。

(2) Mybatis 直接编写原生态 SQL , 可以严格控制 SQL 执行性能 , 灵活度高 , 非常适合对关系数据模型要求不高的软件开发 , 因为这类软件需求变化频繁 , 一但需求变化要求迅速输出成果。但是灵活的前提是 Mybatis 无法做到数据库无关性 , 如果需要实现支持多种数据库的软件 , 则需要自定义多套 SQL 映射文件 , 工作量大。

(3) Hibernate 对象/关系映射能力强 , 数据库无关性好 , 对于关系模型要求高的软件 , 如果用 Hibernate 开发可以节省很多代码 , 提高效率。

6.6 #{}和\${}的区别是什么 ?

#{}是预编译处理 , \${}是字符串替换。

Mybatis 在处理 #{} 时 , 会将 sql 中的 #{} 替换为 ? 号 , 调用 PreparedStatement 的 set 方法来赋值 ;

Mybatis 在处理 \${} 时 , 就是把 \${} 替换成变量的值。

使用 #{} 可以有效的防止 SQL 注入 , 提高系统安全性。

6.7 Mybatis 是如何进行分页的 ? 分页插件的原理是什么 ?

Mybatis 使用 RowBounds 对象进行分页 , 它是针对 ResultSet 结果集执行的内存分页 , 而非物理分页。可以在 SQL 内直接书写带有物理分页的参数来完成物理分页功能 , 也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用 Mybatis 提供的插件接口 , 实现自定义插件 , 在插件的拦截方法内拦截待执行的 SQL , 然后重写 SQL , 根据 dialect 方言 , 添加对应的物理分页语句和物理分页参数。

6.8 Mybatis 是如何将 SQL 执行结果封装为目标对象并返回的？都有哪些映射形式？

第一种是使用<resultMap>标签，逐一定义数据库列名和对象属性名之间的映射关系。

第二种是使用<resultType>标签和 SQL 列的别名功能，将列的别名书写为对象属性名。

有了列名与属性名的映射关系后，Mybatis 通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

6.9 Mybatis 动态 SQL 有什么用？执行原理？有哪些动态 sql？

Mybatis 动态 SQL 可以在 Xml 映射文件内，以标签的形式编写动态 sql，执行原理是根据表达式的值 完成逻辑判断并动态拼接 sql 的功能。

Mybatis 提供了 9 种动态 sql 标签：trim | where | set | foreach | if | choose | when | otherwise | bind。

6.10 Xml 映射文件中，除了常见的 select|insert|update|delete 标签之外，还有哪些标签？

<resultMap>.<parameterMap>.<sql>.<include>.<selectKey>，加上动态 sql 的 9 个标签，其中<sql>为 sql 片段标签，通过<include>标签引入 sql 片段，<selectKey>为不支持自增的主键生成策略标签。

6.11 Mybatis 的 Xml 映射文件中，不同的 Xml 映射文件，id 是否可以重复？

不同的 Xml 映射文件，如果配置了 namespace，那么 id 可以重复；如果没有配置 namespace，那么 id 不能重复；

原因就是 namespace+id 是作为 Map<String, MapperStatement>的 key 使用的，如果没有 namespace，就剩下 id，那么，id 重复会导致数据互相覆盖。有了 namespace，自然 id 就可以重复，namespace 不同，namespace+id 自然也就不同。

6.12 为什么说 Mybatis 是半自动 ORM 映射工具？它与全自动的区别在哪里？

Hibernate 属于全自动 ORM 映射工具，使用 Hibernate 查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而 Mybatis 在查询关联对象或关联集合对象时，需要手动编写 sql 来完成，所以，称之为半自动 ORM 映射工具。

6.13 MyBatis 实现一对一有几种方式？具体怎么操作的？

有联合查询和嵌套查询，联合查询是几个表联合查询，只查询一次，通过在 resultMap 里面配置 association 节点配置一对一的类就可以完成；

嵌套查询是先查一个表，根据这个表里面的结果的外键 id，去再另外一个表里面查询数据，也是通过 association 配置，但另外一个表的查询通过 select 属性配置。

6.14 MyBatis 实现一对多有几种方式，怎么操作的？

有联合查询和嵌套查询。联合查询是几个表联合查询，只查询一次，通过在 resultMap 里面的 collection 节点配置一对多的类就可以完成；嵌套查询是先

查一个表,根据这个表里面的 结果的外键 id,去再另外一个表里面查询数据,也是通过配置 collection,但另外一个表的查询通过 select 节点配置。

6.15 Mybatis 是否支持延迟加载？如果支持，它的实现原理是什么？

答 :Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association 指的就是一对一，collection 指的就是一对多查询。在 Mybatis 配置文件中，可以配置是否启用延迟加载
`lazyLoadingEnabled=true|false`。

它的原理是，使用 CGLIB 创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用 `a.getB().getName()`，拦截器 `invoke()`方法发现 `a.getB()`是 null 值，那么就会单独发送事先保存好的查询关联 B 对象的 sql，把 B 查询上来，然后调用 `a.setB(b)`，于是 a 的对象 b 属性就有值了，接着完成 `a.getB().getName()`方法的调用。这就是延迟加载的基本原理。

当然了，不光是 Mybatis，几乎所有的包括 Hibernate，支持延迟加载的原理都是一样的。

6.16 Mybatis 的一级.二级缓存

1) 一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，**默认打开一级缓存且不能关闭**。

2) 二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，**要手动开启二级缓存**，使用

二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态),可在它的映射文件中配置 <cache/> ；

3) 对于缓存数据更新机制,当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行了 C/U/D 操作后,默认该作用域下所有 select 中的缓存将被 clear。

6.17 什么是 MyBatis 的接口绑定? 有哪些实现方式?

接口绑定,就是在 MyBatis 中任意定义接口,然后把接口里面的方法和 SQL 语句绑定,我们直接调用接口方法就可以,这样比起原来 SqlSession 提供的方法我们可以有更加灵活的选择和设置。

接口绑定有两种实现方式,一种是通过注解绑定,就是在接口的方法上面加上 @Select.@Update 等注解,里面包含 Sql 语句来绑定;另外一种就是通过 xml 里面写 SQL 来绑定,在这种情况下,要指定 xml 映射文件里面的 namespace 必须为接口的全路径名。当 Sql 语句比较简单时候,用注解绑定,当 SQL 语句比较复杂时候,用 xml 绑定,一般用 xml 绑定的比较多。

6.18 使用 MyBatis 的 mapper 接口调用时有哪些要求?

Mapper 接口方法名和 mapper.xml 中定义每个 sql 的 id 相同;

Mapper 接口方法的输入参数类型和 mapper.xml 中定义每个 sql 的 parameterType 的类型相同;

Mapper 接口方法的输出参数类型和 mapper.xml 中定义每个 sql 的 resultType 的类型相同;

Mapper.xml 文件中的 namespace 即是 mapper 接口的类路径。

6.19 简述 Mybatis 的插件运行原理，以及如何编写一个插件。

Mybatis 仅可以编写针对

ParameterHandler.ResultSetHandler.StatementHandler.Executor 这 4 种接口的插件，

在四大对象创建的时候

1、每个创建出来的对象不是直接返回的，而是

`interceptorChain.pluginAll(parameterHandler);`

2、获取到所有的 Interceptor（拦截器）（插件需要实现的接口）；

调用 `interceptor.plugin(target);` 返回 target 包装后的对象

3、插件机制，我们可以使用插件为目标对象创建一个代理对象；AOP（面向切面）

我们的插件可以为四大对象创建出代理对象；

代理对象就可以拦截到四大对象的每一个执行；

Mybatis 使用 JDK 的动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 `InvocationHandler` 的 `invoke()` 方法，当然，只会拦截那些你指定需要拦截的方法。

编写插件：实现 Mybatis 的 Interceptor 接口并复写 intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

七. SpringBoot SpringCloud 框架

7.1 传统开发一般是基于什么框架进行开发的？

传统开发开发方式我们一般采用 SSH 或者 SSM 进行开发。随着互联网的发展和社会业务场景的复杂性增加。为了能够更加高效快捷的开发，我们慢慢趋于使用 SpringBoot 进行开发。但针对一些传统行业(比如：银行项目，政府项目，或者比较保密的项目我们还是使用传统，因为他们使用时间更久更稳定。)

SSH 通常指的是 Struts2 做控制器(controller)，spring 管理各层的组件，hibernate 负责持久化层。

SSM 则指的是 SpringMVC 做控制器(controller)，Spring 管理各层的组件，MyBatis 负责持久化层。

共同点 :1.Spring 依赖注入 DI来管理各层的组件。2.使用面向切面编程 AOP 管理事物.日志.权限等。

不同点：1.Struts2 和 SpringMVC 控制器(controller)控制视图和模型的交互机制的不同，

Struts2 是 Action 类级别，SpringMVC 是方法级别，更容易实现 RESTful 风格。

7.2 什么是 SOA 的？

SOA，对于刚接触企业的工作的朋友来说，可能不大了解这个概念。下面就通过一些讲解来跟大家简单分享一下 SOA。SOA，面向服务的体系结构（Service-Oriented Architecture）。它的主要作用是将应用程序的不同功能单元（称为服务）通过这些服务之间定义良好的接口和契约联系起来。

接口是采用中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。

简而言之：SOA 是一种微服务的架构，是一种架构思想，把我们项目中涉及业务模块做更小的拆分，降低耦合，一遍满足客户更多，更广的需求。基于他的思想我们一般会引入 Dubbo 消息中间件，或者 SpringCloud 框架。

7.3 什么是微服务？

7.3.1 微服务的概念

微服务架构风格是一种将单个应用程序作为一套小型服务开发的方法，每种应用程序都在自己的进程中运行，并与轻量级机制（通常是 HTTP 资源 API）进行通信。这些服务是围绕业务功能构建的，可以通过全自动部署机制独立部署。这些服务的集中管理最少，可以用不同的编程语言编写，并使用不同的数据存储技术。

简而言之：微服务是把我们要做的项目根据业务进行独立部署，可以把同一项目中不同语言编写的业务基于 Restful 风格开发接口进行网络间通讯。

7.3.2 微服务的特点

1. 独立部署，灵活扩展

传统的单体架构是以整个系统为单位进行部署，而微服务则是以每一个独立组件（例如用户服务，商品服务）为单位进行部署。什么意思呢？比如根据每个服务的吞吐量不同，支付服务需要部署 20 台机器，用户服务需要部署 30 台机器，而商品服务只需要部署 10 台机器。这种灵活部署只有微服务架构才能实现。而近几年流行的 Docker，为微服务架构提供了有效的容器。

2. 资源的有效隔离

微服务设计的原则之一，就是每一个微服务拥有独立的数据源，假如微服务 A 想要读写微服务 B 的数据库，只能调用微服务 B 对外暴露的接口来完成。这样有效避免了服务之间争用数据库和缓存资源所带来的问题。

3. 团队组织架构的调整

微服务设计的思想也改变了原有的企业研发团队组织架构。传统的研发组织架构是水平架构，前端有前端的团队，后端有后端的团队，DBA 有 DBA 的团队，测试有测试的团队。而微服务的设计思想对团队的划分有着一定的影响，使得团队组织架构的划分更倾向于垂直架构，比如用户业务是一个团队来负责，支付业务是一个团队来负责。



7.3.3 为什么要使用微服务架构，微服务的优缺点是什么？

单一架构模式在项目初期的时候开发、测试、部署方便，但是随着项目逐步开发，项目工程会很大，最终互相之间会有繁琐的 jar 包。

1. 不再适用于敏捷开发，任何开发人员都不能完全理解，在修复漏洞和添加新功能时候变得复杂

2. 项目模块越大，启动越慢，很小的改动也需要重新部署整个项目

3. 任何模块的漏洞，都会降低系统的可靠性

4. 如果想整体使用新的技术，新的框架，那是不可能的

如果采用微服务，解决了单一系统的复杂性，主要有以下优点

1. 可采用敏捷开发模式，选择合适的技术

2. 每一个微服务是独立部署的，可以快速迭代部署

3. 一些需要负载均衡的服务可以使用 nginx 同一反向代理分发请求，这样不需要整个系统负载均衡了

微服务的缺点：

1. 微服务作为分布式系统时候带来了复杂性，当应用是整体应用时，模块之间可以内部调用，但微服务是多个独一的应用，调用起来增加了一定的复杂性

2. 多个数据库，事物的实现会头痛

3. 测试微服务变得复杂，当一个服务依赖于另一个服务时候，测试时需要另一个服务的支持

4. 整体系统只需要部署在一组相同的服务器上，而微服务需要更高的自动化形似

7.3.4 SOA 和微服务的区别？

SOA（面向服务的架构）：面向服务的架构（SOA）是一个组件模型，它将应用程序的不同功能单元（称为服务）通过这些服务之间定义良好的接口和契约联系起来。接口是采用中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。这使得构建在各种各样的系统中的服务可以以一种统一和通用的方式进行交互。

微服务：微服务架构是一种将单个应用程序作为一套小型服务开发的方法，每种应用程序都在自己的进程中运行，并与轻量级机制（通常是 HTTP 资源 API）进行通信。这些服务是围绕业务功能构建的，可以通过全自动部署机制独立部署。这些服务的集中管理最少，可以用不同的编程语言编写，并使用不同的数据存储技术。

SOA 和微服务的主要区别：

微服务剔除 SOA 中复杂的 ESB 企业服务总线，所有的业务智能逻辑在服务内部处理，使用 Http（Rest API）进行轻量化通讯

SOA 强调按水平架构划分为：前、后端、数据库、测试等，微服务强调按垂直架构划分，按业务能力划分，每个服务完成一种特定的功能，服务即产品

SOA 将组件以 library 的方式和应用部署在同一个进程中运行，微服务则是各个服务独立运行。

传统应用倾向于使用统一的技术平台来解决所有问题，微服务可以针对不同业务特征选择不同技术平台，去中心统一化，发挥各种技术平台的特长。

SOA 架构强调的是异构系统之间的通信和解耦合；（一种粗粒度、松耦合的服务架构）

微服务架构强调的是系统按业务边界做细粒度的拆分和部署。

7.4 SpringBoot 是什么？为什么我们选择使用 SpringBoot 开发？

分析：由于以上的分析，xml 的可读性差，基于注解的方式也有不足，那这时候，为了解决这些问题，更加的提升开发效率 SpringBoot 登场了。它的到来 就把我们的 xml 配置基本上都省略了。但是也不是绝对的完全省略，而是让我们操作应用起来跟简单了。但是它也有

一个弊端，那就是一旦出了问题不好定位。

什么是 SpringBoot？

简单的说，SpringBoot 就是整合了很多优秀的框架，不用我们自己手动的去写一堆 xml 配置然后进行配置。

从本质上来说，SpringBoot 就是 Spring，它做了那些没有它你也会去做的 Spring Bean 配置。它使用“习惯优于配置”（项目中存在大量的配置，此外还内置了一个习惯性的配置，让你无需手动进行配置）的理念让你的项目快速运行起来。使用 SpringBoot 很容易创建一个独立运行（运行 jar，内嵌 Servlet 容器）准生产级别的基于 Spring 框架的项目，使用 SpringBoot 你可以不用或者只需要很少的 Spring 配置。

为什么要用 SpringBoot？（敏捷开发）

快速的完成 Spring 和其他框架的构建整合。进一步提升开发效率。这就是当今流行的敏捷开发。

7.4.1 SpringBoot 的发家史之前情回顾

1. 回顾 java 中 “配置” 的发展历史（引出 SpringBoot）

什么是配置文件

.ini..properties.*.xml 都是配置文件，这些配置文件往往被奉若神明，自从我们接触电脑的那一天开始，就被人警告不要乱弄他们。其实他们与 java 的 HashMap 结构是一样的，都是一个存放 key-value 对的容器，只是配置文件那些是属性与属性值的 key-value 对罢了。但是为何不能乱碰它呢？因为这些 key-valueyi 对一般是记录着程序运行的参数，而且很多是初始化的参数，一般这些参数都是自程序的开始运行，也就是程序的出生到程序被关闭，也就是程序死亡都需要使用的定值。正如我们每一个人的名字.性别.出生地这些属性一样，从我们出生到死亡都要不停地用。

java 中最原始的时候没有任何配置，以及配置的发展历程

概述：在 java 开发的婴儿期，不需要考虑配置，因为程序很简单，在编码过程中就可以完成几乎所有的需求。

第一阶段： *.properties *.xml 配置文件

随着业务的增强，和代码复杂度的提高，这时候一些公用的部分，以及项目初始化到项目卸载的的运行过程的一些必备参数都抽取到配置文件当中。以上在 spring1.x 的时代被广泛应用。

第二阶段： 注解配置

伴随着更加复杂的业务，以及代码代码的相互依赖的特性，这时候，庞大的配置文件成了我们头疼的事情。就在这时候，迎着 spring2.x 时代的到来，也随着 JDK 发行 1.5 版本对注解的支持，这时候我们就用注解代替之前的 xml 配置。

第三阶段： java 配置（基于类的配置）

又随着时间的推移，在 spring3.x 的到来，为我们提供了更简单的配置，那就是基于 java 的配置。这样一来 我们连 spring 核心配置文件都省了，只需要在你的类上加一个 @Configuration 注解，就代表当前类是一个可配置的类，接着就可以用 @Bean 来配置当前方法返回的是一个 bean 实例对象。

总结：综上所述 在几种配置方式中，可以对比一下它们各自的优势和短板。

xml 容易操控，对于一些经常修改的配置更适合，但是可读性差开发过程中一定程度上限制了开发效率。

注解 是一种分散式的元数据，与源代码紧绑定。配置简单，易操作，但是由于注解是分散式的，所以维护起来不易。

java 配置 java 配置其实就是注解的有一种拓展，目的是为了更加的简单易用，更加的减少 xml 的配置。

7.4.2 SpringBoot 环境搭建

步骤分析：

建项目

创建一个 maven 工程

注意 SpringBoot 整合的时候在建立 maven 工程的时候一定要创建一个父工程 而且，父工程选择 2.x 的版本，因为我们用的是 spring5.x 的版本。

导依赖

spring-boot-web.jar

写配置... 配置 SpringBoot 的核心配置文件 application.properties

此处省略一堆字。因为我们今天要用 SpringBoot

创建类

在 src 目录下的 com.itihaimaba 包下新建一个 SpringBootApplicationRunner 的类。并且在类上加上 @SpringBootApplication 的注解来表示此类就是一个 SpringBoot 的启动器。另外还需要我们在此类中写一个 main 方法，方法中利用 SpringApplication.run (SpringBootApplicationRunner.class,args) ; 来完成最终的启动。

访问 localhost : 8080

分析 spring-boot-web 这个依赖：由于这个依赖的导入，根据 maven 的依赖机制他给我们导入了一大堆支持我们开发的依赖。

7.4.3 SpringMVC 的传统环境搭建

注：在进行 SpringBoot 和 SpringMVC 整合之前咱们先回顾一下传统的 SpringMVC

环境搭建

步骤分析：

1. 建项目

新建 maven 项目 SpringMVCfirst

注意 这时候要选择 war 包

2. 导依赖

servlet-api

jsp-api

spring-context

spring-webmvc

3. 写配置

**** 配置 web.xml 文件**

配置前端核心控制器 DispatcherServlet

```
<serverlet>
```

```
    <serverlet-name>DispatcherServlet</serverlet-name>
```

```
    <serverlet-class>类路径</serverlet-class>
```

```
    <init-param>
```

```
        <param-name>contextConfigLocation</param-name>
```

```
        <param-value>classpath:SpringMVC.xml</param-value>
```

```
    </init-param>
```

```
    <load-on-stratup>1</load-onstratup>
```

```
</serverlet>
```

```
<serverlet-mapping>
```

```
    <serverlet-name>DispatcherServlet</serverlet-name>
```

```
    <url-pattern>/</ url-pattern >
```

```
</serverlet-mapping>
```

****配置 SpringMVC.xml**

写入约束文件

```
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
xmlns:mvc="http://www.springframework.org/schema/mvc"
```

```
xmlns:context="http://www.springframework.org/schema/context"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="
```

```
http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans.xsd
```

```
http://www.springframework.org/schema/mvc
```

```
http://www.springframework.org/schema/mvc/spring-mvc.xsd
```

```
http://www.springframework.org/schema/context
```

```
http://www.springframework.org/schema/context/spring-context.xsd">
```

配置要扫描的包

```
<context-component-scan
```

```
base-pagkage=" ..." ></context-component-scan>
```

配置视图解析器

```
<bean id=" viewResolver" class=" InternalResourceViewResolver" >
```

```
-- 前缀
```

```
<property name=" prefix" value=" /WEB-INF/pages/" ></property>
```

```
-- 后缀
```

```
<property name=" suffix" value=" .jsp" ></property>
```

```
</bean>
```

开启 spring 对注解 mvc 的支持

<mvc:annotation-driven/>

4. 创建类

** 在 src 目录下创建 com.itheima.web 包，然后创建 TestController

** 在 webapp 目录下新建 index.jsp

** 在 WEB-INF 下创建 pages 文件夹，然后创建一个 success.jsp

5. 将项目 add 到 Tomcat 下运行 然后访问测试 !!!

7.4.4 SpringBoot 整合 SpringMVC 的环境搭建

入门案例：SpringBoot 整合 Spring 和 SpringMVC

步骤分析：

** 在 com.itheima 包下新建一个 web.controller 包

** 在 web.controller 下新建 HelloController

** 加入 @RestController 注解就 ok 啦！

** 然后通过访问 localhost:8080/...

解释：

当 SpringBoot 工程启动的时候，就把我们请求对象和相应对象都封装到 Jackson 的 jar 包当中了。它的请求和响应都是利用 json 传输的。

7.4.5 SpringBoot 原理分析及启动时的一些细节

1. 原理分析：

解释此类的作用：

这个类就是我们 SpringBoot 整合其他框架的一个启动类，一会所有准备都完事后，只要一运行此类，那么整合就完事了。

解释@SpringBootApplication 的作用：

@SpringBootApplication = @Configuration +
@EnableAutoConfiguration + @ComponentScan

@Configuration -- 提到@Configuration 就要提到他的搭档@Bean。使用这两个注解就可以创建一个简单的 spring 配置类，可以用来替代相应的 xml 配置文件。

@EnableAutoConfiguration -- 能够自动配置 spring 的上下文，试图猜测和配置你想要的 bean 类，通常会自动根据你的类路径和你的 bean 定义自动配置。

@ComponentScan -- 会自动扫描指定包下的全部标有@Component 的类，并注册成 bean

经常见到的几个 JDK 的元注解：

@Target(ElementType.TYPE) // 当前注解的使用范围

@Retention(RetentionPolicy.RUNTIME) //生命周期

@Documented // 声明在生成 doc 文档时是否带着注解

@Inherited // 声明是否子类会显示父类的注解

总结：一个@SpringBootApplication 注解就搞定了所有事，因为它觉得以上三个注解都是我们固定的配置，所以就给我们省略了。又封装出一个注解来代替，这就是 SpringBoot 的核心设计思想。

思考--我们为什么要把 SpringBoot 工程的启动类创建在 base 包下？

SpringBoot 工程运行时，扫描包来创建容器会扫描启动类所在的包以及它下面的子包。

1. 细节一分析

`SpringApplication.run ( SpringApplicationRunner.class,args ) ;`

备注：

run 方法：

此方法是用于把启动类的字节码传入给 SpringBoot 的核心容器，通过注解配置解析来创建 IOC 容器。

参数一：就是启动类的字节码。

参数二：通常情况下就是 main 方法的 args

**拓展：args 由内容吗？

验证：进行打印，会出现一个对象的地址

解释：数组默认情况下会打印它的第一个对象的地址。

2. 细节二分析

备注：我们关注控制台，有一个 SpringBoot 的图标。那这个图标我们可以不让他打印吗？

解释：可以。按照以下操作

```
SpringBootApplication sba = new SpringApplication(class);
```

```
sba.setBannerMode(Mode.off);
```

```
sba.run(args);
```

7.4.6 SpringBoot 热部署的使用

1. 重启传统的 SpringMVC 项目，访问一把！OK

修改：此时随便在 controller 类中改改 再访问！不 OK

原因：我们改变源码了，但是没有重新部署 !!!

解决：重新部署一遍，又 OK 了 !!!

2. 分析：

问题：总是重新部署势必会花费大量的时间！这时候大胆畅想一下能不能修改源码后不重新部署也能生效呢？

注意：修改源码必须重新编译，重启是重新编译的手段。这个思想是没有问题的，但是我们此时要想的是能不能不重新部署也能完成重新编译 !!!

解决：利用 SpringBoot 的热部署就可以实现。

3. 如何实现 SpringBoot 的热部署

-- 导入依赖

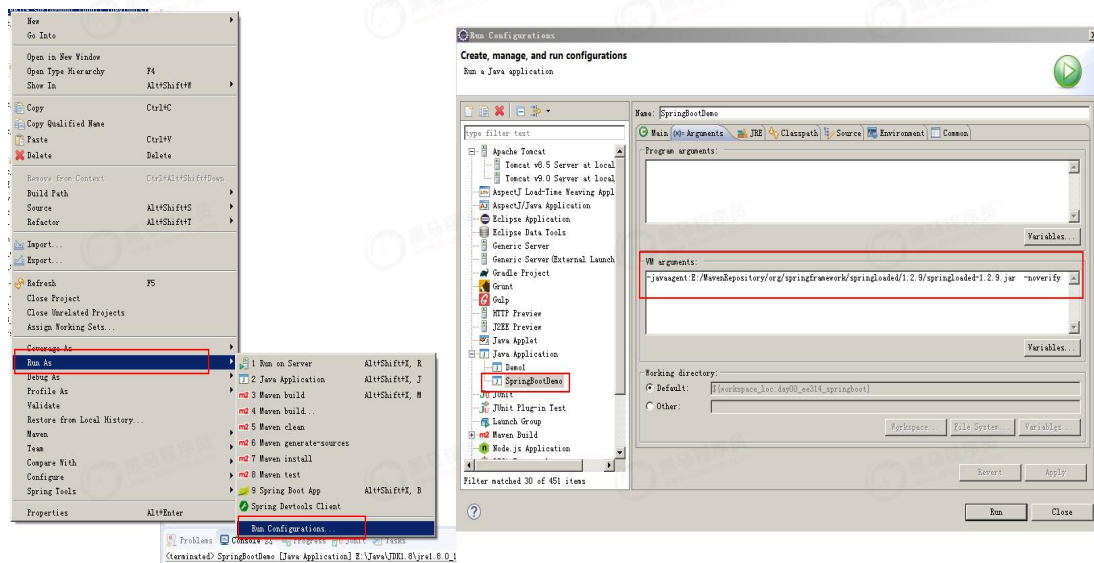
spring-boot-devtools.jar 2.0.0

导完坐标保存后我们的项目在开发工具中会有一个变化

```
_01springboot [boot] [devtools]
```

springloaded-1.2.9.jar

注意：这个 jar 包我们直接导入不了，需要手动改变一下版本号。导入包以后还需要我们 在 eclipse 中配置一下如下图：



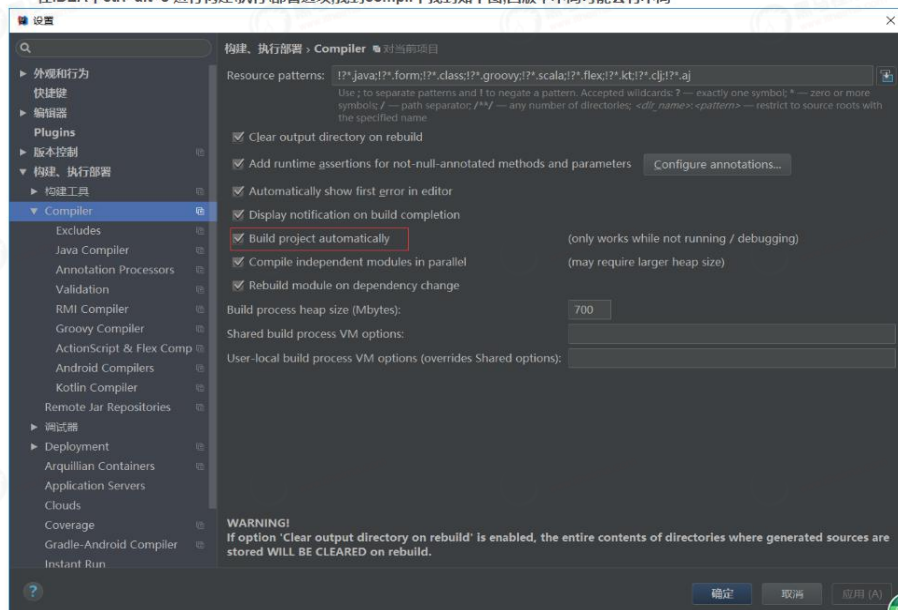
参数配置：是 springloaded-1.2.9.jar 这个包在 maven 本地仓库的位置。

此时热部署就可以正常使用了...

注意：在 IDEA 中热部署需要设置一下，步骤如下：

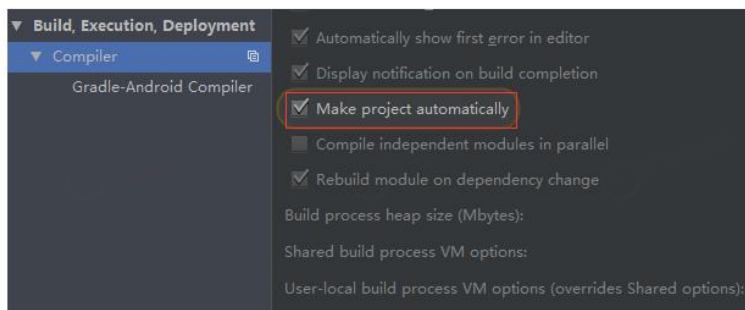
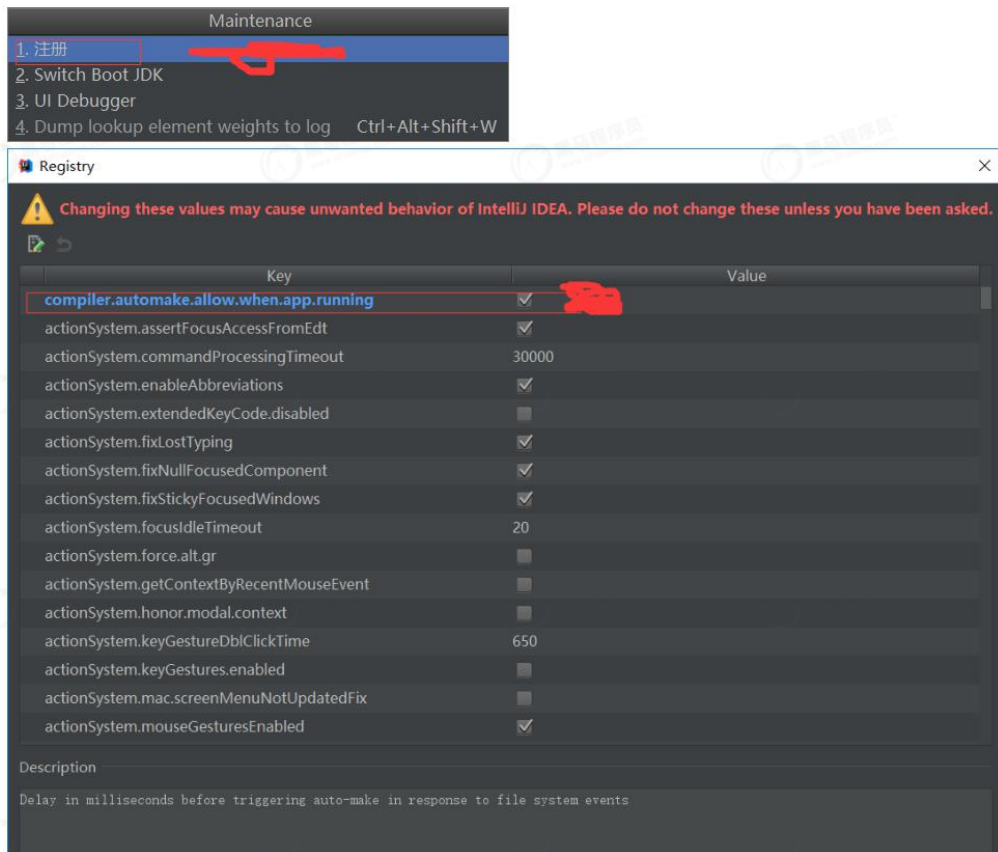
第一步：

在IDEA中ctrl+alt+s 进行构建执行部署选项,找到compil中找到如下图,因版本不同可能会有不同



第二步：

这时进入maven的pom文件,按下快捷键 `ctrl+shift+alt+/,`找到registry(注册),勾选下图内容



最后重启 IDEA，导入热部署的 依赖 jar 包即可！

7.4.7 SpringBoot 热部署涉之 springloaded1.2.9jar 包的安装

1. 特别强调！针对 springloaded.jar,由于 maven 中央仓库目前

最高版是 1.2.8 没有 1.2.9 那我们如何获取这个包呢？

第一步：把 springloaded-1.2.9.放到你电脑的任意一个没有中文和空格的目录下路径下。

第二步：然后通过 cmd 命令窗口进入 刚才新建的文件夹下。

第三步： 执行 maven 安装命令

```
mvn install:install-file
```

```
-DgroupId=org.springframework -DartifactId=springloaded
```

```
-Dversion=1.2.9
```

```
-Dpackaging=jar
```

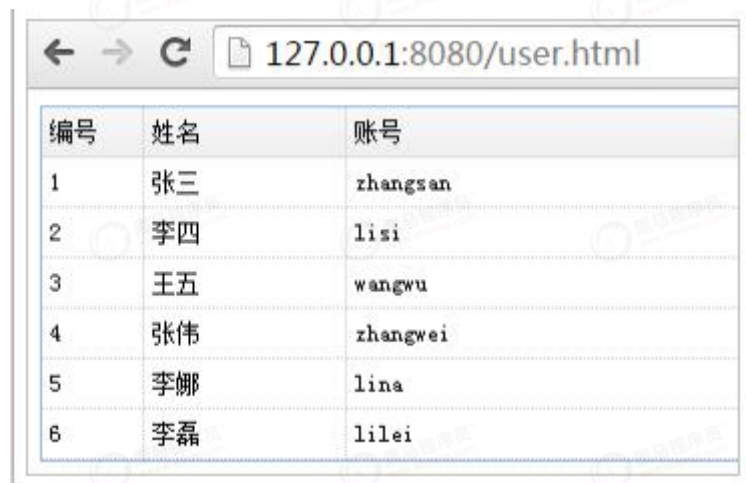
```
-Dfile=springloaded-1.2.9.BUILD.jar
```

7.4.8 SpringBoot 整合 SpringData JPA

1. 需求分析：

使用 SpringBoot + Spring MVC + Spring Data JPA + EasyUI

框架组合实现部门列表查询，效果如下：



编号	姓名	账号
1	张三	zhangsan
2	李四	lisi
3	王五	wangwu
4	张伟	zhangwei
5	李娜	lina
6	李磊	lilei

2. 步骤分析：

在数据库中创建 user 表并添加相关数据

在 MySQL 数据库执行以下语句:

```
DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `user_name` varchar(100) DEFAULT NULL COMMENT '用户名',
  `password` varchar(100) DEFAULT NULL COMMENT '密码',
  `name` varchar(100) DEFAULT NULL COMMENT '姓名',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT
CHARSET=utf8;

INSERT INTO `user` VALUES ('1', 'zhangsan', '123456', '张三');
INSERT INTO `user` VALUES ('2', 'lisi', '123456', '李四');
INSERT INTO `user` VALUES ('3', 'wangwu', '123456', '王五');
INSERT INTO `user` VALUES ('4', 'zhangwei', '123456', '张伟');
INSERT INTO `user` VALUES ('5', 'lina', '123456', '李娜');
INSERT INTO `user` VALUES ('6', 'lilei', '123456', '李磊');
```

创建一个新的项目

在刚才整合完 SpringMVC 的基础上进行整合

导依赖

MySQL 驱动包

spring-boot-starter-data-jpa.jar

写配置

在项目的 resources 目录下新建 application.properties 文件

此文件中配置一些 经常修改 的信息，此处我们配置的是数据库连接信息。

创建类

** 创建实体类 并利用注解建立与数据库的对应关系。

** 创建 Dao 接口并且 extends JpaRepository 和

JpaSpecificationExecutor

** 创建 service 层

** 创建 Controller 层，写相应的查询方法

7.4.9 SpringBoot 项目,前端使用 JQuery EasyUI

引入 jQueryeasyUI 插件，将数据显示到表格中

步骤分析：

在 resources 目录下新建一个 **static 文件夹**，必须命名为 static 这是

SpringBoot 的规定，然后将 easyUI 的相关文件放入，最后直接访问 user.html

通过 ajax 请求请求后台数据。

7.4.10 SpringBoot 整合 MyBatis

步骤分析：

1. 建项目（还用刚才的）

2. 导依赖

在 pom.xml 中加入以下依赖

<!-- SpringBoot 的 Mybatis 启动器 -->

```
<dependency>
```

```
    <groupId>org.mybatis.spring.boot</groupId>
```

```
    <artifactId>mybatis-spring-boot-starter</artifactId>
```

```
    <version>1.1.1</version>
```

```
</dependency>
```

3. 写配置 (SpringBoot 的核心配置文件，此时不用改！)

4. 创建类

新建 IUserMapper 接口 写一个根据 username 模糊查

询 User 集合的接口

和之前的方式一样，只是多了两个注解：

@Mapper：声明 Mapper 接口

@Select：声明这个接口所需要使用的 sql，当然，有查询的注解，肯定就

有增删改的注解。

```
import java.util.List;
```

```
import org.apache.ibatis.annotations.Mapper;
```

```
import org.apache.ibatis.annotations.Select;
```

```
import cn.itcast.info.pojo.User;
```

```
@Mapper
```

```
public interface UserMapper {
```

```
@Select("select * from user where name like  
'%${value}%'")
```

```
public List<User> queryUserByName(String name);  
}
```

Service 层新增一个接口和实现方法

进入 controller 新增一个方法（此时需要传入参数，我们用一下 REST 风格）

```
@RequestMapping( "findUserByName/{name}" )  
public List<User> findUserByName(@PathVariable( "name" )String  
name){  
  
    List<User> list = userService.findUserByName(name);  
    Return list;    }  

```

总结：

通过以上操作我们发现一个问题，就是热部署也不是万能的，当我们修改了 Dao 层中注解里面的 SQL 语句并不能自动部署。这个问题要注意，它属于一个细节，但是往往细节也能决定你的身价！！

7.4.11 SpringBoot 整合 Redis

1. 准备工作

Windows 下开启 Redis 服务

为了方便查看，开启 Redis 的一个客户端管理工具

2. 整合步骤分析

建项目（此处还用原来的项目）

导依赖

在 pom.xml 加入依赖

```
<!-- 配置使用 Redis 启动器 -->
```

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-Redis</artifactId>
```

```
</dependency>
```

写配置（注意：SpringBoot 的核心配置不用改，但是此时由于是要缓存数据库，需要配置若干注解）

打开 SpringBoot 启动类，加入@EnableCaching 意思就是开启缓存的支持。

接着到 service 中在配置缓存的具体适用规则。

```
** @Cacheable(value=" ..." )
```

注：在执行查询的方法上加@Cacheable 注解表示第一次查询到数据库取值然后在缓存中存一份，当再次查询的时候就先到缓存中取值，知道执行缓存清理操作缓存数据才消失，其中重点分析注解中的两个参数 value：指定存入 Redis 中的 key 的值。

```
**@CacheEvict(value=" ..." )
```

注：这个注解适用于清除缓存，里面的参数和存入注解时的一样，只是多一个 allEntries 属性表示是否彻底清除。

7.4.12 SpringBoot 整合 Junit

1. 步骤分析：

建项目（此处还用以上的项目为例）

导依赖

在 pom.xml 中加入测试依赖

```
<! 配置测试启动器 >
```

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-test</artifactId>
```

```
<scope>test</scope>
```

```
</dependency>
```

写配置（SpringBoot 核心配置不用改）

创建类

在工程的 test 目录下新建一个测试类

测试类内容如下：

```
//替换运行器
```

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```
//指定引导类的字节码
```

```
@SpringBootTest(classes=SpringBootApplicationRunner.class)
```

```
public class SpringBootTest {
```

```
@Autowired
```

```
private IUserService userService;

@Test

public void testFindAll() {

    List<User> users = userService.findAll();

    for(User user : users) {

        System.out.println(user);

    }

}

}
```

总结：SpringBoot 的 Junit 整合其实和 spring 对 Junit 的整合差不多，只是换了一个注解配置。

7.4.13 SpringBoot 读取 application.properties 配置文件

1. properties 文件通常是做什么的？

在我们程序中，一些我们经常要进行修改的属性，通常配在 properties 文件中，这样当修改的时候就可以修改 properties 文件就可以了。但是有一个环节必不可少，那就是我们想用这些属性的时候，必然要去读取 properties 文件。所以读取 properties 文件也是我们程序中必须的一部分。

2. 在 SpringBoot 中如何读取核心配置文件 application.properties

步骤分析：在测试类中注入一个 Environment 对象。private Environment environment;然后调用这个对象的 getProperty() 方法就可以了。

```
environment.getProperty(key);
```

注意：此对象只用于读取 SpringBoot 的核心配置文件，不是它核心配置文件的内容读不出来。

7.4.14 SpringBoot 读取自定义 properties 文件

1. 需求分析：

自定义一个 mail.properties 文件

再创建一个 MailProperty.java 对象

mail.properties 文件中的值和 MailProperty.java 对象的属性对应。然后最终直接操作 MailProperty.java 对象就相当于操作 mail.properties 文件，从而达到不直接读取 mail.properties 文件就能对其进行操作的目的。

2. 操作步骤：

自定义一个 mail.properties 文件，再创建一个 MailProperty.java 对象
mail.properties 文件中的值和 MailProperty.java 对象的属性对应。在 MailProperty.java 对象上配置两个注解

```
@ConfigurationProperties(prefix=" mail" )
```

```
@PropertySource(value=" classpath:mail.properties" )
```

注意：@ConfigurationProperties prefix 属性的含义就是指定 properties 文件中的属性的前缀。

@PropertySource 注解是指定对应的 properties 文件的位置。

配置完了还不能用，此时我们需要一个 SpringBoot 的一个依赖
spring-boot-configuration-processor

完成以上步骤后，发现还不行，此时我们还需要一个注解的配合才能完成。

@Component 或者 @Configuration

把以上注解配置到我们的 MailProperty.java 上就相当于把这个类当做一个可配置的类，才能够最终将 MailProperty.java 注入到 IOC 容器中。

总结：这样做的好处就是当项目中有大量的 properties 文件的时候我们可以不用频繁的解析配置文件，而是转化成操作对象的方式，变得更简单。

7.4.15 SpringBoot 读取 yml 配置文件

概述：yml 文件也是 SpringBoot 为我们提供的一种核心配置文件的格式，它除了能像 application.properties 文件那样写配置，另外他也提供了一种新的数据结构就是更加 一目了然 的树形结构。

用法：

1. 读取普通数据

使用@Value 注解配合 Spring 的 EL 表达

```
@Value("${mail.host}")
```

```
private String mailHost;
```

2. 读取对象数据

和读取自定义的 properties 一样

7.5 SpringCloud 是什么？

7.5.1 概念：

SpringCloud 是一系列框架的有序集合，它利用 SpringBoot 的开发便利

性简化了分布式系统的开发，比如服务发现.服务网关.服务路由.链路追踪等。其设计目的是为了简化 Spring 应用的搭建和开发过程。该框架遵循“约定大于配置”原则，采用特定的方式进行配置，从而使开发者无需定义大量的 XML 配置。通过这种方式，SpringBoot 致力于在蓬勃发展的快速应用开发领域成为领导者。SpringCloud 并不重复造轮子，而是将市面上开发得比较好的模块集成进去，进行封装，从而减少了各模块的开发成本。换句话说：Spring cloud 提供了构建分布式系统所需的“全家桶”。核心组件:Eureka.Ribbon.Feign.Hystrix.Zuul.

Eureka :各个服务启动时 ,Eureka Client 都会将服务注册到 Eureka Server ,并且 Eureka Client 还可以反过来从 Eureka Server 拉取注册表，从而知道其他服务在哪里

Ribbon : 服务间发起请求的时候，基于 Ribbon 做负载均衡，从一个服务的多台机器中选择一台

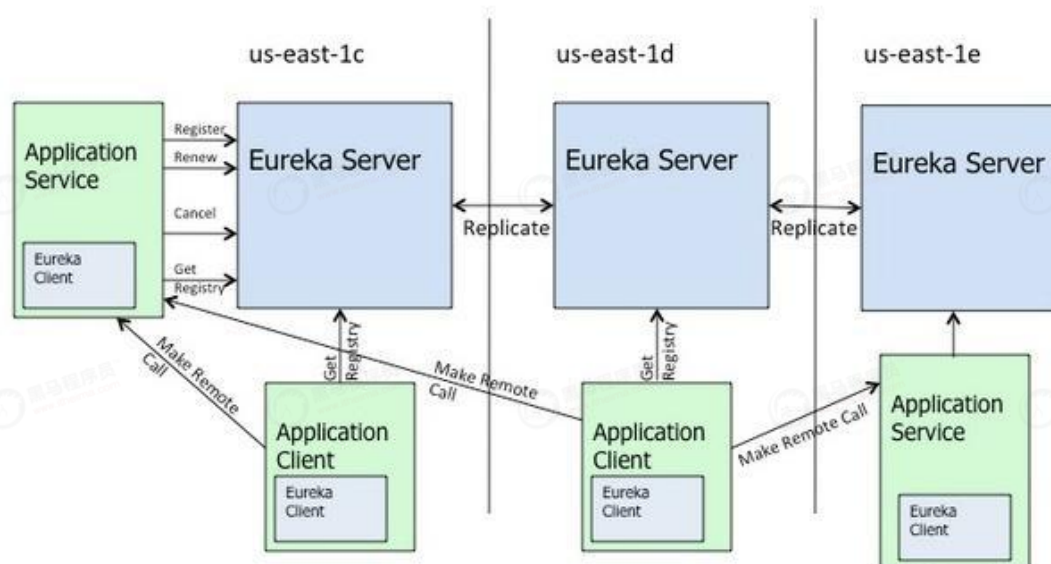
Feign : 基于 Feign 的动态代理机制，根据注解和选择的机器，拼接请求 URL 地址，发起请求

Hystrix : 发起请求是通过 Hystrix 的线程池来走的，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题

Zuul : 如果前端.移动端要调用后端系统，统一从 Zuul 网关进入，由 Zuul 网关转发请求给对应的服务

7.5.2 核心组件讲解：

7.5.2.1 Eureka(服务中心)



作用：实现服务治理（服务注册与发现）

简介：SpringCloud Eureka 是 SpringCloud Netflix 项目下的服务治理模块。

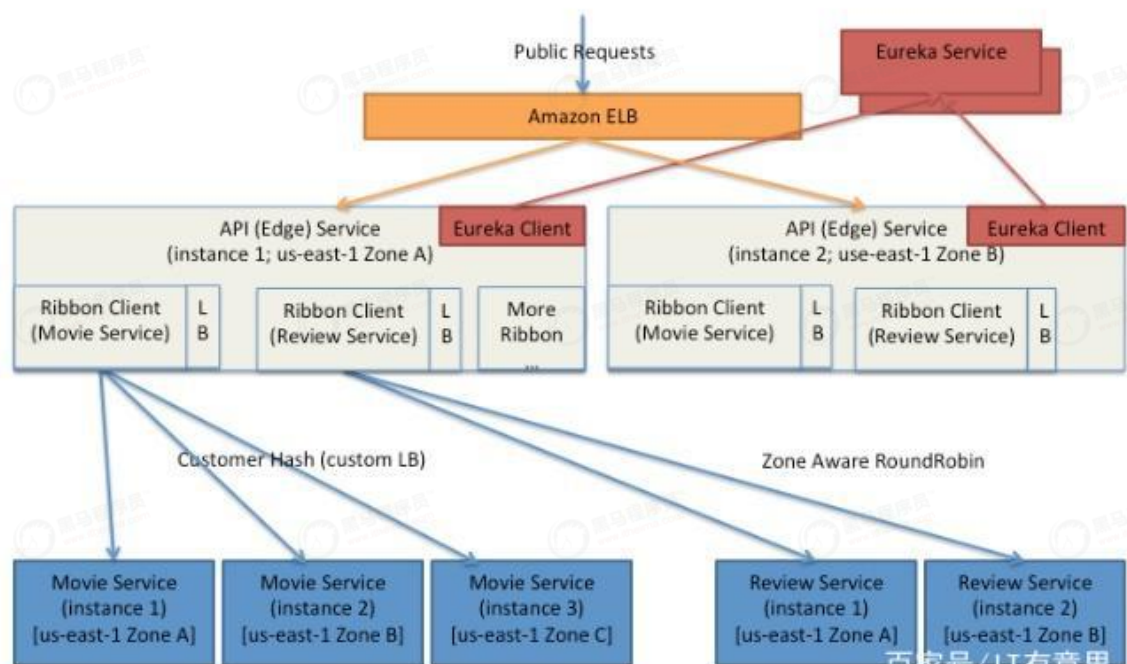
由两个组件组成：Eureka 服务端和 Eureka 客户端。

Eureka 服务端用作服务注册中心。支持集群部署。

Eureka 客户端是一个 java 客户端，用来处理服务注册与发现。

在应用启动时，Eureka 客户端向服务端注册自己的服务信息，同时将服务端的服务信息缓存到本地。客户端会和服务端周期性的进行心跳交互，以更新服务租约和服务信息。

7.5.2.2 Ribbon(负载均衡工具)

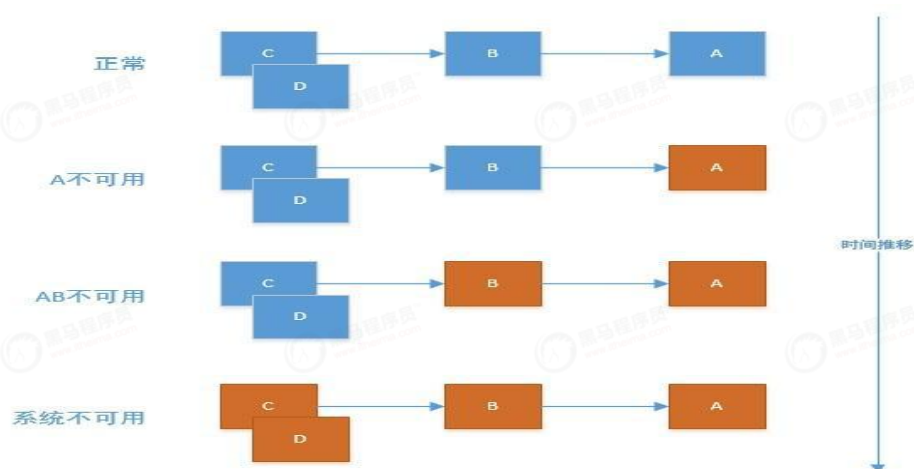


作用：Ribbon，主要提供客户侧的软件负载均衡算法。

简介：SpringCloud Ribbon 是一个基于 HTTP 和 TCP 的客户端负载均衡工具，它基于 Netflix Ribbon 实现。通过 S 的封装，可以让我们轻松地将面向服务的 REST 模版请求自动转换成客户端负载均衡的服务调用。

注意看上图，关键点就是将外界的 rest 调用，根据负载均衡策略转换为微服务调用。Ribbon 有比较多的负载均衡策略，以后专门讲解。

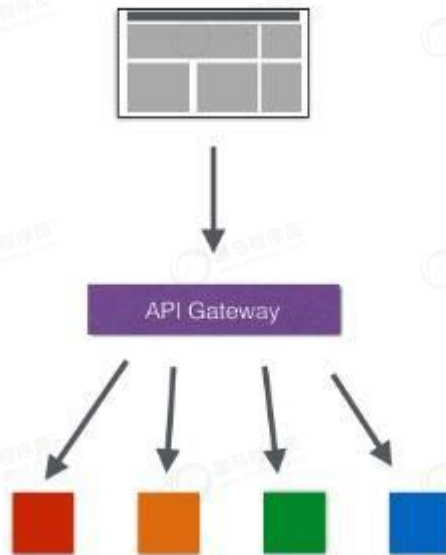
7.5.2.3 Hystrix(熔断器)



作用：断路器，保护系统，控制故障范围。

简介：为了保证其高可用，单个服务通常会集群部署。由于网络原因或者自身的原因，服务并不能保证 100%可用，如果单个服务出现问题，调用这个服务就会出现线程阻塞，此时若有大量的请求涌入，Servlet 容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应。

7.5.2.4 Zuul(服务网关)

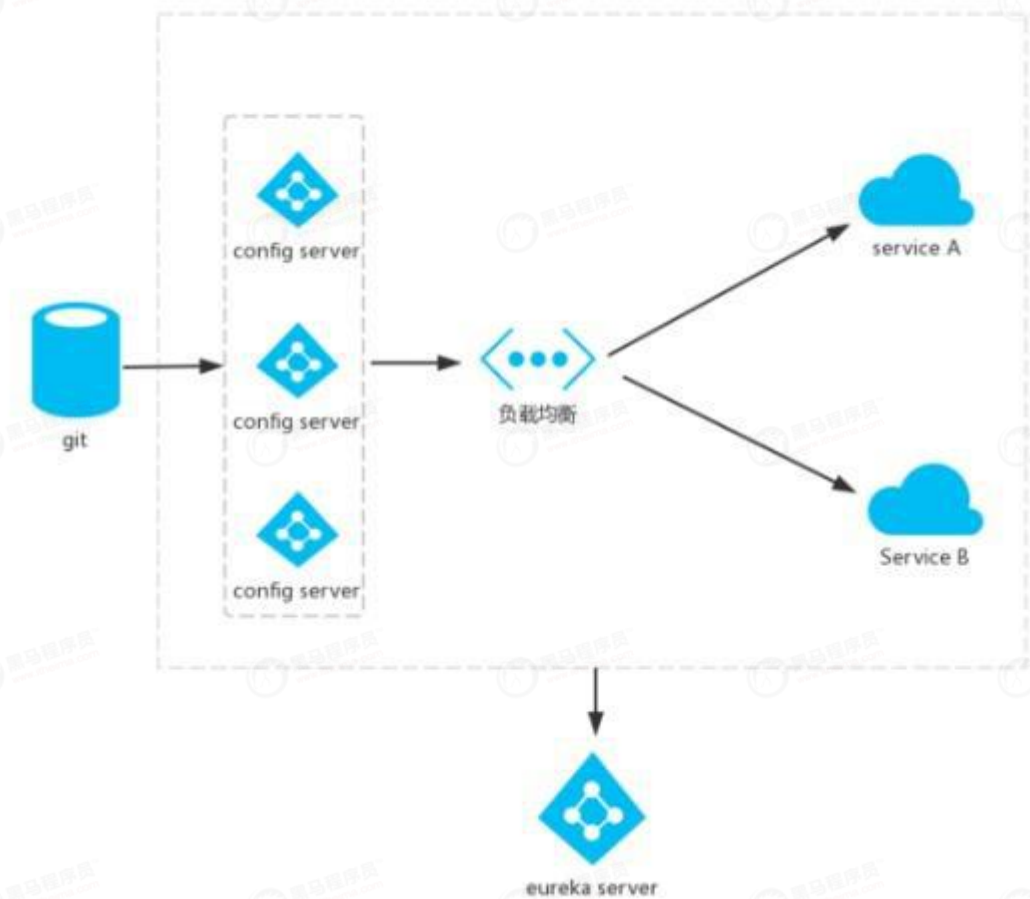


作用：api 网关，路由，负载均衡等多种作用

简介：类似 nginx，反向代理的功能，不过 netflix 自己增加了一些配合其他组件的特性。

在微服务架构中，后端服务往往不直接开放给调用端，而是通过一个 API 网关根据请求的 url，路由到相应的服务。当添加 API 网关后，在第三方调用端和服务提供方之间就创建了一面墙，这面墙直接与调用方通信进行权限控制，后将请求均衡分发给后台服务端。

7.5.2.5 Config(配置中心)



作用：配置管理

简介：SpringCloud Config 提供服务器端和客户端。服务器存储后端的默认实现使用 git，因此它轻松支持标签版本的配置环境，以及可以访问用于管理内容的各种工具。

这个还是静态的，得配合 SpringCloud Bus 实现动态的配置更新。

7.5.3 SpringCloud 基于业务场景对于各个组件的讲解

先来给大家说一个业务场景，假设咱们现在开发一个电商网站，要实现支付订单的功能，流程如下：

创建一个订单之后，如果用户立刻支付了这个订单，我们需要将订单状态更新为“已支付”

扣减相应的商品库存 通知仓储中心，进行发货 给用户的这次购物增加相应的积分

针对上述流程，我们需要有**订单服务.库存服务.仓储服务.积分服务**。整个流程的大体思路如下：

用户针对一个订单完成支付之后，就会去找订单服务，更新订单状态

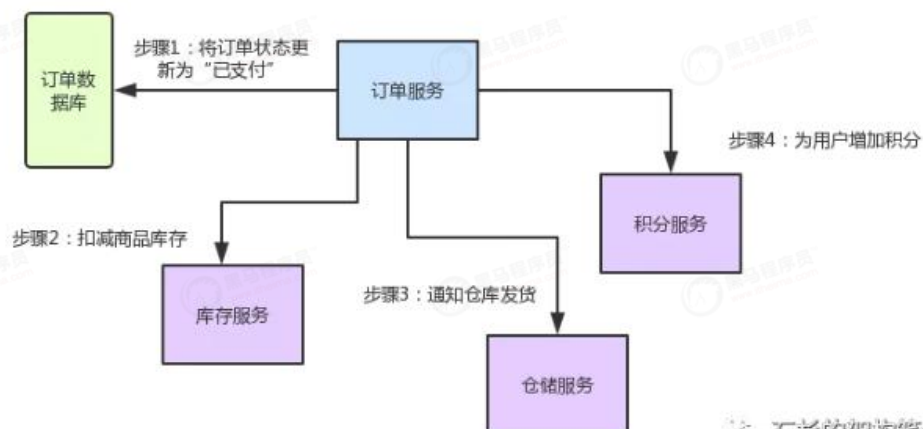
订单服务调用库存服务，完成相应功能

订单服务调用仓储服务，完成相应功能

订单服务调用积分服务，完成相应功能

至此，整个支付订单的业务流程结束

下图这张图，清晰表明了各服务间的调用过程：



好！有了业务场景之后，咱们就一起来看看 Spring Cloud 微服务架构中，这几个组件如何相互协作，各自发挥的作用以及其背后的原理。

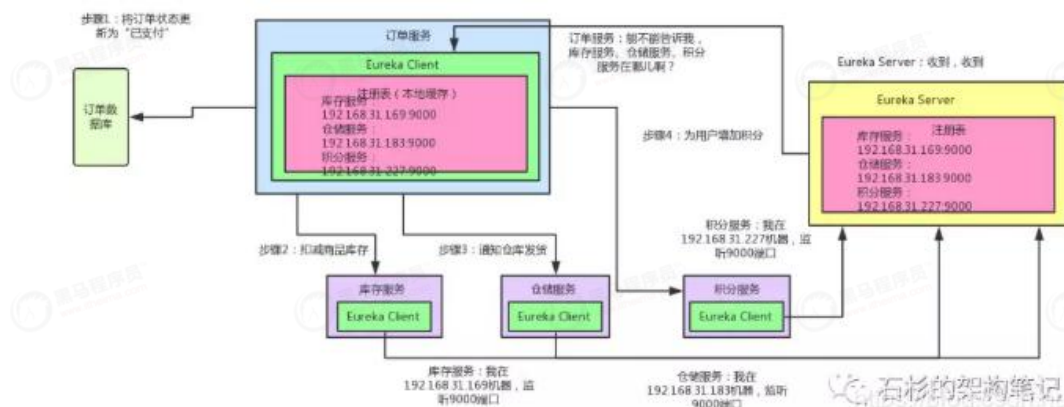
SpringCloud 核心组件：Eureka

咱们来考虑第一个问题：订单服务想要调用库存服务.仓储服务，或者是积分服务，怎么调用？

订单服务压根儿就不知道人家库存服务在哪台机器上啊！他就算想要发起一个请求，都不知道发送给谁，有心无力！

这时候，就轮到 SpringCloud Eureka 出场了。Eureka 是微服务架构中的注册中心，专门负责服务的注册与发现。

咱们来看看下面的这张图，结合图来仔细剖析一下整个流程：



如上图所示，库存服务.仓储服务.积分服务中都有一个 Eureka Client 组件，这个组件专门负责将这个服务的信息注册到 Eureka Server 中。说白了，就是告诉 Eureka Server，自己在哪台机器上，监听着哪个端口。而 Eureka Server 是一个注册中心，里面有一个注册表，保存了各服务所在的机器和端口号

订单服务里也有一个 Eureka Client 组件，这个 Eureka Client 组件会找 Eureka Server 问一下：库存服务在哪台机器啊？监听着哪个端口啊？仓储服务

呢？积分服务呢？然后就可以把这些相关信息从 Eureka Server 的注册表中拉取到自己本地缓存起来。

这时如果订单服务想要调用库存服务，不就可以找自己本地的 Eureka Client 问一下库存服务在哪台机器？监听哪个端口吗？收到响应后，紧接着就可以发送一个请求过去，调用库存服务扣减库存的那个接口！同理，如果订单服务要调用仓储服务、积分服务，也是如法炮制。

总结一下：

Eureka Client：负责将这个服务的信息注册到 Eureka Server 中

Eureka Server：注册中心，里面有一个注册表，保存了各个服务所在的机器和端口号

SpringCloud 核心组件：Feign

现在订单服务确实知道库存服务、积分服务、仓库服务在哪里了，同时也监听着哪些端口号了。但是新问题又来了：**难道订单服务要自己写一大堆代码，跟其他服务建立网络连接，然后构造一个复杂的请求，接着发送请求过去，最后对返回的响应结果再写一大堆代码来处理吗？**

这是上述流程翻译的代码片段，咱们一起来看看，体会一下这种绝望而无助的感受！！！！

友情提示，前方高能：

```

1 CloseableHttpClient httpClient = HttpClients.createDefault();
2 HttpPost httpPost = new HttpPost("http://192.168.31.169:9000/");
3
4 List<NameValuePair> parameters = new ArrayList<NameValuePair>();
5 parameters.add(new BasicNameValuePair("scope", "project"));
6 parameters.add(new BasicNameValuePair("q", "java"));
7
8 UrlEncodedFormEntity formEntity = new UrlEncodedFormEntity(parameters);
9 httpPost.setEntity(formEntity);
10 httpPost.setHeader(
11     "User-Agent",
12     "Mozilla/5.0 (Windows NT 6.3; Win64; x64)"
13 );
14
15 CloseableHttpResponse response = null;
16 response = httpClient.execute(httpPost);
17 if (response.getStatusLine().getStatusCode() == 200) {
18     String content = EntityUtils.toString(response.getEntity(), "UTF-8");
19     System.out.println(content);
20 }
21 if (response != null) {
22     response.close();
23 }
24 httpClient.close();

```

石杉的架构笔记
https://blog.csdn.net/forezp

看完上面那一大段代码，有没有感到后背发凉，一身冷汗？实际上你进行服务间调用时，如果每次都手写代码，代码量比上面那段要多至少几倍，所以这个事儿压根儿就不是地球人能干的。

既然如此，那怎么办呢？别急，Feign 早已为我们提供好了优雅的解决方案。来看看如果用 Feign 的话，你的订单服务调用库存服务的代码会变成啥样？

```

1 @FeignClient("inventory-service")
2 public class InventoryService {
3     @RequestMapping(value = "/reduceStock/{goodsSkuId}", method = HttpMethod.PUT)
4     @ResponseBody
5     public ResultCode reduceStock(@PathVariable("goodsSkuId") Long goodsSkuId);
6 }
7
8 @Service
9 public class OrderService {
10     @Autowired
11     private InventoryService inventoryService;
12
13     public ResultCode payOrder() {
14         // 步骤1: 更新本地数据库订单状态为“已支付”
15         orderDAO.updateStatus(id, OrderStatus.PAYED);
16         // 步骤2: 调用库存服务，扣减商品库存
17         inventoryService.reduceStock(goodsSkuId);
18     }
19 }

```

石杉的架构笔记
https://blog.csdn.net/forezp

看完上面的代码什么感觉？是不是感觉整个世界都干净了，又找到了活下去的勇气！没有底层的建立连接、构造请求、解析响应的代码，直接就是用注解定义一个 FeignClient 接口，然后调用那个接口就可以了。人家 Feign Client 会在

底层根据你的注解,跟你指定的服务建立连接.构造请求.发起请求.获取响应.解析响应,等等。这一系列脏活累活,人家 Feign 全给你干了。

那么问题来了,Feign 是如何做到这么神奇的呢?很简单,**Feign 的一个关键机制就是使用了动态代理**。咱们一起来看看下面的图,结合图来分析:

首先,如果你对某个接口定义了@FeignClient 注解,Feign 就会针对这个接口创建一个动态代理

接着你要是调用那个接口,本质就是会调用 Feign 创建的动态代理,这是核心中的核心

Feign 的动态代理会根据你在接口上的@RequestMapping 等注解,来动态构造出你要请求的服务的地址

最后针对这个地址,发起请求.解析响应



SpringCloud 核心组件: Ribbon

说完了 Feign, 还没完。现在新的问题又来了, 如果人家库存服务部署在了 5 台机器上, 如下所示:

192.168.169:9000

192.168.170:9000

192.168.171:9000

192.168.172:9000

192.168.173:9000

这下麻烦了！人家 Feign 怎么知道该请求哪台机器呢？

这时 SpringCloud Ribbon 就派上用场了。Ribbon 就是专门解决这个问题的。它的作用是负载均衡，会帮你在每次请求时选择一台机器，均匀的把请求分发到各个机器上

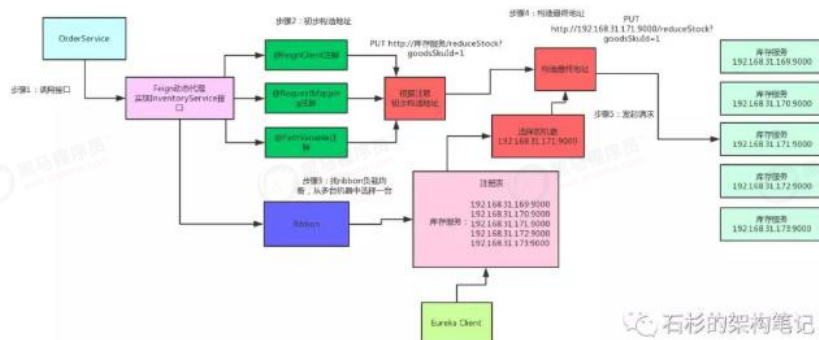
Ribbon 的负载均衡默认使用的最经典的 Round Robin 轮询算法。这是啥？简单来说，就是如果订单服务对库存服务发起 10 次请求，那就先让你请求第 1 台机器.然后是第 2 台机器.第 3 台机器.第 4 台机器.第 5 台机器，接着再来一个循环，第 1 台机器.第 2 台机器。。。以此类推。

此外，Ribbon 是和 Feign 以及 Eureka 紧密协作，完成工作的，具体如下：

首先 Ribbon 会从 Eureka Client 里获取到对应的服务注册表，也就知道了所有的服务都部署在了哪些机器上，在监听哪些端口号。

然后 Ribbon 就可以使用默认的 Round Robin 算法，从中选择一台机器 Feign 就会针对这台机器，构造并发起请求。

对上述整个过程，再来一张图，帮助大家更深刻的理解：



SpringCloud 核心组件：Hystrix

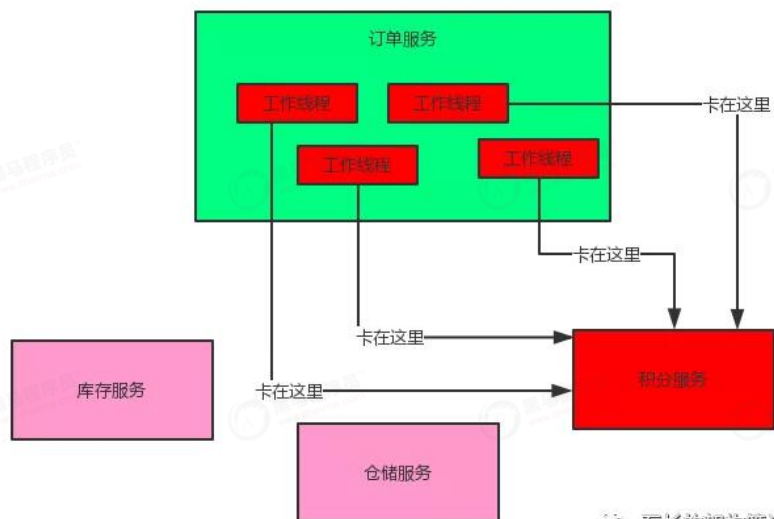
在微服务架构里，一个系统会有很多的服务。以本文的业务场景为例：订单服务在一个业务流程里需要调用三个服务。现在假设订单服务自己最多只有 100 个线程可以处理请求，然后呢，积分服务不幸的挂了，每次订单服务调用积分服务的时候，都会卡住几秒钟，然后抛出一个超时异常。

咱们一起来分析一下，这样会导致什么问题？

如果系统处于高并发的场景下，大量请求涌过来的时候，订单服务的 100 个线程都会卡在请求积分服务这块。导致订单服务没有一个线程可以处理请求

然后就会导致别人请求订单服务的时候，发现订单服务也挂了，不响应任何请求了

上面这个，就是微服务架构中恐怖的服务雪崩问题，如下图所示：



如上图，这么多服务互相调用，要是不做任何保护的话，某一个服务挂了，就会引起连锁反应，导致别的服务也挂。比如积分服务挂了，会导致订单服务的线程全部卡在请求积分服务这里，没有一个线程可以工作，瞬间导致订单服务也挂了，别人请求订单服务全部会卡住，无法响应。

但是我们思考一下,就算积分服务挂了,订单服务也可以不用挂啊!为什么?

我们结合业务来看:支付订单的时候,只要把库存扣减了,然后通知仓库发货就 OK 了

如果积分服务挂了,大不了等他恢复之后,慢慢人肉手工恢复数据!为啥一定要因为一个积分服务挂了,就直接导致订单服务也挂了呢?不可以接受!

现在问题分析完了,如何解决?

这时就轮到 Hystrix 闪亮登场了。Hystrix 是隔离.熔断以及降级的一个框架。啥意思呢?说白了, Hystrix 会搞很多个小小的线程池,比如订单服务请求库存服务是一个线程池,请求仓储服务是一个线程池,请求积分服务是一个线程池。每个线程池里的线程就仅仅用于请求那个服务。

打个比方:现在很不幸,积分服务挂了,会咋样?

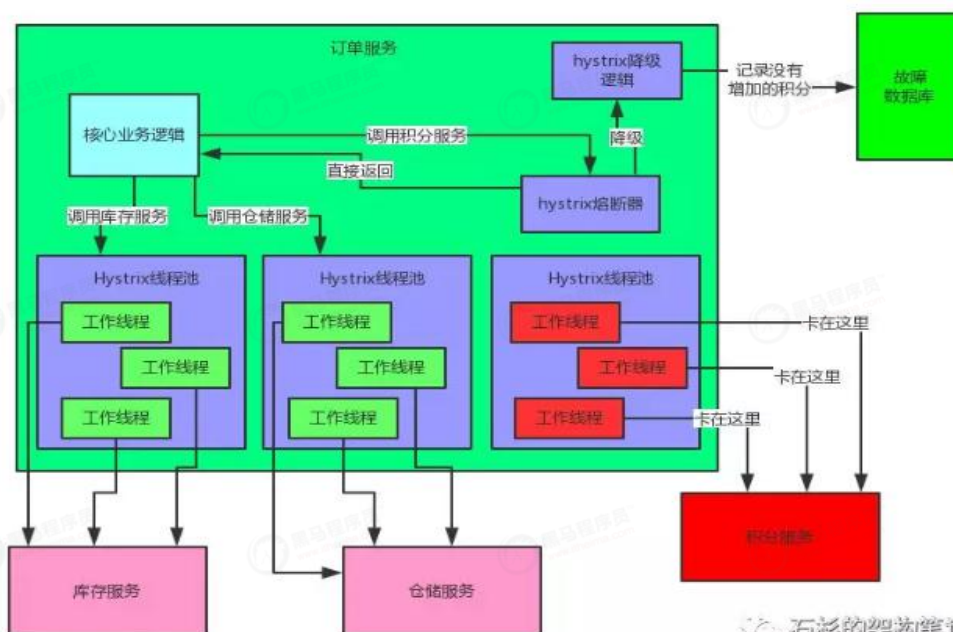
当然会导致订单服务里的那个用来调用积分服务的线程都卡死不能工作了啊!但是由于订单服务调用库存服务.仓储服务的这两个线程池都是正常工作的,所以这两个服务不会受到任何影响。

这个时候如果别人请求订单服务,订单服务还是可以正常调用库存服务扣减库存,调用仓储服务通知发货。只不过调用积分服务的时候,每次都会报错。**但是如果积分服务都挂了,每次调用都要去卡住几秒钟干啥呢?有意义吗?当然没有!**所以我们直接对积分服务熔断不就得了,比如在 5 分钟内请求积分服务直接就返回了,不要去走网络请求卡住几秒钟,这个过程,就是所谓的熔断!

那人家又说,兄弟,积分服务挂了你就熔断,好歹你干点儿什么啊!别啥都不干就直接返回啊?没问题,咱们就来个降级:每次调用积分服务,你就在数据库里记录一条消息,说给某某用户增加了多少积分,因为积分服务挂了,导致没

增加成功！这样等积分服务恢复了，你可以根据这些记录手工加一下积分。这个过程，就是所谓的降级。

为帮助大家更直观的理解，接下来用一张图，梳理一下 Hystrix 隔离.熔断和降级的全流程：



SpringCloud 核心组件：Zuul

说完了 Hystrix，接着给大家说说最后一个组件：Zuul，也就是微服务网关。

这个组件是负责网络路由的。不懂网络路由？行，那我给你说说，如果没有 Zuul 的日常工作会怎样？

假设你后台部署了几百个服务，现在有个前端兄弟，人家请求是直接从浏览器那儿发过来的。**打个比方**：人家要请求一下库存服务，你难道还让人家记着这服务的名字叫做 inventory-service？部署在 5 台机器上？就算人家肯记住这一个，你后台可有几百个服务的名称和地址呢？难不成人家请求一个，就得记住一个？你要这样玩儿，那真是友谊的小船，说翻就翻！

上面这种情况，压根儿是不现实的。所以一般微服务架构中都必然会设计一个网关在里面，像 android.ios.pc 前端.微信小程序.H5 等等，不用去关心后端有几百个服务，就知道有一个网关，所有请求都往网关走，网关会根据请求中的一些特征，将请求转发给后端的各个服务。

而且有一个网关之后，还有很多好处，比如可以做统一的降级.限流.认证授权.安全，等等。

7.6 SpringBoot 常见面试问题总结

7.6.1 什么是 SpringBoot

1. 用来简化 spring 应用的初始搭建以及开发过程 使用特定的方式来进行配置 (properties 或 yml 文件)
2. 创建独立的 spring 引用程序 main 方法运行
3. 嵌入的 Tomcat 无需部署 war 文件
4. 简化 maven 配置
5. 自动配置 spring 添加对应功能 starter 自动化配置

7.6.2 SpringBoot 常用的 starter 有哪些

- 1.spring-boot-starter-web (嵌入 tomcat 和 web 开发需要 servlet 与 jsp 支持)
- 2.spring-boot-starter-data-jpa (数据库支持)
- 3.spring-boot-starter-data-Redis (Redis 数据库支持)
- 4.spring-boot-starter-data-solr (solr 搜索应用框架支持)
- 5.mybatis-spring-boot-starter (第三方的 mybatis 集成 starter)

7.6.3 SpringBoot 自动配置的原理

1. @EnableAutoConfiguration 这个注释告诉 SpringBoot “猜” 你将如何想配置 Spring, 基于你已经添加 jar 依赖项。如果 spring-boot-starter-web 已经添加 Tomcat 和 Spring MVC, 这个注释自动将假设您正在开发一个 web 应用程序并添加相应的 spring 设置。会自动去 maven 中读取每个 starter 中的 spring.factories 文件 该文件里配置了所有需要被创建 spring 容器中的 bean

2. 使用 @SpringBootApplication 注解 可以解决根类或者配置类 (我自己的说法, 就是 main 所在类) 头上注解过多的问题, 一个 @SpringBootApplication 相当于 @Configuration, @EnableAutoConfiguration 和 @ComponentScan 并具有他们的默认属性值

7.6.4 SpringBoot 如何添加【修改代码】自动重启功能

添加开发者工具集====spring-boot-devtools

7.6.5 SpringBoot 的核心配置文有哪几个? 它们的区别是什么?

SpringBoot 的核心配置文文件是 application 和 bootstrap 配置文文件。 application 配置文文件这个容易理解, 主要用用于 Spring boot 项目目的自自动化配置。 bootstrap 配置文文件有以下几个应用用场景。使用 SpringCloud Config 配置中心时 SpringBoot, 这时需要在 bootstrap

配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；一些固定的不能被覆盖的属性；一些加密/解密场景；具体请看这篇文章《核心配置文件详解》。

7.6.6 SpringBoot 的配置文件有哪几种格式？它们有什么区别？

.properties 和 .yml，它们的区别主要是书写格式不同。1).properties
app.user.name = javastack 2).yml 1. app: 2. user: 3. name:
javastack 另外，.yml 格式不支持 @PropertySource 注解导入配置。

SpringBoot 的核心注解是哪个？它主要由哪几个注解组成的？

启动类上面的注解是@SpringBootApplication，它也是 SpringBoot 的核心注解，主要组合包含了以下 3 个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文
件的功能。@EnableAutoConfiguration：打开自动配置的功能，也可以
关闭某个自动配置的选项，如关闭数据源自动配置功能：

@SpringBootApplication(exclude = {
DataSourceAutoConfiguration.class })。@ComponentScan：Spring
组件扫描。

7.6.7 运行 SpringBoot 有哪几种方式？

1) 打包用命令或者放到容器中运行

2) 用 Maven/ Gradle 插件运行

3)直接执行行行 main 方方法运行

7.6.8 如何在 SpringBoot 启动的时候运行一些特定的代码？

可以实现接口口 ApplicationRunner 或者 CommandLineRunner ,这两个 接口口实现方方式一一样，它们都只提供了了一个 run 方方法，具体请看这篇文文 章《SpringBoot Runner 启动器器》

7.6.9 SpringBoot 有哪几种读取配置的方式？

SpringBoot 可以通过 @PropertySource,@Value,@Environment, @ConfigurationProperties 来绑定变量量 ,具体请看这篇文文章《SpringBoot 读取配置的几几种方方式》

7.6.10 SpringBoot 支持哪些日志框架？推荐和默认的日志框架是哪个？

SpringBoot 支支持 Java Util Logging, Log4j2, Lockback 作为日志框 架,如果你使用用 Starters 启动器器 ,SpringBoot 将使用用 Logback 作为 默认日志框架，具体请看这篇文文章《SpringBoot日志集成》。

7.6.11 SpringBoot 实现热部署有哪几种方式？

主要有两种方式： Spring Loaded Spring-boot-devtools
Spring-boot-devtools 使用用方方式可以参考这篇文文章《SpringBoot 实现热部署》。

7.6.12 你如何理解 SpringBoot 配置加载顺序？

在 SpringBoot 里里里面面，可以使用用以下几几种方式来加载配置。

- 1) properties文文件；
- 2) YAML文文件；
- 3) 系统环境变量量；
- 4) 命令行行参数； 等等.....
- 5) 具体请看这篇文文章《SpringBoot 配置加载顺序详解》。

7.6.13 SpringBoot 如何定义多套不同环境配置？

提供多套配置文文件，如：

1. application.properties
 2. application-dev.properties
 3. application-test.properties
 4. application-prod.properties 运行行时指定具体的配置文文件，
- 具体请看这篇文文章《SpringBoot Profile 不不同环境配置》。

7.6.14 SpringBoot 可以兼容老 Spring 项目吗 ,如何做？

可以兼容 ,使用用 @ImportResource 注解导入入老老老 Spring 项目目配置文文 件。

7.6.15 保护 SpringBoot 应用有哪些方法？

在生产中使用 HTTPS 使用 Snyk 检查你的依赖关系 升级到最新版本
启用 CSRF 保护 使用内容安全策略防止 XSS 攻击 ... 更多请看这篇文章
《10 种保护 SpringBoot 应用的绝佳方法》

7.6.16 SpringBoot 2.X 有什么新特性？与 1.X 有什么区别？

配置变更 JDK 版本升级 第三方类库升级 响应式 Spring 编程支持
HTTP/2 支持 配置属性绑定 更多改进与加强... 具体请看这篇文章
《SpringBoot 2.x 新特性总结及迁移指南》

7.6.17 SpringBoot 自动配置原理是什么？

注解 @EnableAutoConfiguration, @Configuration,
@ConditionalOnClass 就是自动配置的核心, 首先它得是一个配置文
件, 其次根据类路径下是否有这个类去自动配置。 具体看这篇文章
《SpringBoot 自动配置原理. 实战》

7.7 SpringCloud 常见面试问题总结

7.7.1 什么是 SpringCloud？

SpringCloud 流应用程序启动器是基于 SpringBoot 的 Spring 集成应用程序,
提供与外部系统的集成。SpringCloud Task, 一个生命周期短暂的微服务
框架, 用于快速构建执行有限数据处理的应用程序。

7.7.2 使用 SpringCloud 有什么优势？

使用 SpringBoot 开发分布式微服务时，我们面临以下问题

- 1.与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。
- 2.服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。
- 3.冗余-分布式系统中的冗余问题。
- 4.负载均衡 负载均衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。
- 5.性能-问题 由于各种运营开销导致的性能问题。
- 6.部署复杂性-Devops 技能的要求。

7.7.3 服务注册和发现是什么意思？SpringCloud 如何实现？

当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。Eureka 服务注册和发现可以在这种情况下提供帮助。由于所有服务都在 Eureka 服务器上注册并通过调用 Eureka 服务器完成查找，因此无需处理服务地点的任何更改和处理

7.7.4 负载均衡的意义什么？

在计算中，负载均衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载均衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载均衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载均衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

7.7.5 什么是 Hystrix？它如何实现容错？

Hystrix 是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。

通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。

思考以下微服务

假设如果上图中的微服务 9 失败了，那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。

随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达 1000。这是 hystrix 出现的地方，我们将使用 Hystrix 在这种情况下的 Fallback 方法功能。我们有两个服务 employee-consumer 使用由 employee-producer 公开的服务。

现在假设由于某种原因，employee-producer 公开的服务会抛出异常。我们在这种情况下使用 Hystrix 定义了一个回退方法。这种后备方法应该具有与公

开服务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值。

7.7.6 什么是 Hystrix 断路器？我们需要它吗？

由于某些原因，employee-consumer 公开服务会引发异常。在这种情况下使用 Hystrix 我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。

如果 firstPage method() 中的异常继续发生，则 Hystrix 电路将中断，并且员工使用者将一起跳过 firstPage 方法，并直接调用回退方法。断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会。

7.7.7 什么是 Netflix Feign？它的优点是什么？

Feign 是受到 Retrofit，JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。Feign 的第一个目标是将约束分母的复杂性统一到 http apis，而不考虑其稳定性。在 employee-consumer 的例子中，我们使用了 employee-producer 使用 REST 模板公开的 REST 服务。

但是我们必须编写大量代码才能执行以下步骤

- 1.使用功能区进行负载平衡。
- 2.获取服务实例，然后获取基本 URL。
- 3.利用 REST 模板来使用服务。

前面的代码如下

@Controller

public class ConsumerControllerClient {

@Autowired

private LoadBalancerClient loadBalancer;

public void getEmployee() throws RestClientException, IOException {

 ServiceInstance

 serviceInstance=loadBalancer.choose("employee-producer");

 System.out.println(serviceInstance.getUri());

 String baseUrl=serviceInstance.getUri().toString();

 baseUrl=baseUrl+"/employee";

 RestTemplate restTemplate = new RestTemplate();

 ResponseEntity<String> response=null;

 try{

 response=restTemplate.exchange(baseUrl,

 HttpMethod.GET, getHeaders(),String.class);

 }catch (Exception ex)

 {

 System.out.println(ex);

 }

 System.out.println(response.getBody());

}

之前的代码，有像 NullPointerException 这样的例外的机会，并不是最优的。我们将

看到如何使用 Netflix Feign 使呼叫变得更加轻松和简洁。如果 Netflix Ribbon 依赖关系也在类路径中，那么 Feign 默认也会负责负载均衡。

7.7.8 什么是 SpringCloud Bus ? 我们需要它吗 ?

考虑以下情况：我们有多多个应用程序使用 SpringCloud Config 读取属性，而 Spring cloud Config 从 GIT 读取这些属性。

下面的例子中多个员工生产者模块从 Employee Config Module 获取 Eureka 注册的财产。

如果假设 GIT 中的 Eureka 注册属性更改为指向另一台 Eureka 服务器，会发生什么情况。在这种情况下，我们将不得不重新启动服务以获取更新的属性。

还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调用这个 url。例如，如果 Employee Producer1 部署在端口 8080 上，则调用 `http://localhost:8080/refresh`。同样对于 Employee Producer2 `http://localhost:8081/refresh` 等等。这又很麻烦。这就是 SpringCloud Bus 发挥作用的地方。

SpringCloud Bus 提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新 Employee Producer1，则会自动刷新所有其他必需的模块。如果我们有多多个微服务启动并运行，这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的刷新。

八. SaaS 项目

1. 名词与概念

- SaaS (Software-as-a-Service), 即软件即服务。提供给消费者完整的软件解决方案, 你可以从软件服务商处以租用或购买等方式获取软件应用, 组织用户即可通过 Internet 连接到该应用 (通常使用 Web 浏览器)。所有基础结构、中间件、应用软件和应用程序数据都位于服务提供商的数据中心内。服务提供商负责管理硬件和软件, 并根据适当的服务协议确保应用和数据的可可用性和安全性。SaaS 让组织能够通过最低前期成本的应用快速建成投产。
- SaaS 软件就适用对象而言, 可以划分为针对个人的与针对企业的面面向个人的 SaaS 产品: 在线文档, 账务管理, 文件管理, 日程计划、照片管理、联系人管理, 等等云类型的服务。而面向企业的 SaaS 产品主要包括: CRM (客户关系管理)、ERP (企业资源计划管理)、线上视频或者与群组通话会议、HRM (人力资源管理)、OA(办公系统)、外勤管理、财务管理、审批管理等。
- SaaS 与传统软件相比, a.能够降低企业成本: 按需购买, 即租即用, 无需关注软件的开发维护。b.软件更新迭代快速: 和传统软件相比, 由于 saas 部署在云端, 使得软件的更新迭代速度加快。c.支持远程办公: 将数据存储到云后, 用户即可通过任何连接到 Internet 的计算机或移动设备访问其信息。
- PaaS (Platform-as-a-Service), 即平台即服务。提供给消费者的服务是把客户采用提供的开发语言和工具 (例如 Java , python, .Net 等) 开发的或收购的应用程序部署到供应商的云计算基础设施上去。客户不需要

管理或控制底层的云基础设施，包括网络.服务器.操作系统.存储等，但客户能控制部署的应用程序，也可能控制运行应用程序的托管环境配置。

- IaaS (Infrastructure as a Service)，即基础设施即服务。提供给消费者的服务是对所有计算基础设施的利用，包括处理 CPU.内存.存储.网络和其它基本的计算资源，用户能够部署和运行任意软件，包括操作系统和应用程序。消费者不管理或控制任何云计算基础设施，但能控制操作系统的选择.存储空间.部署的应用，也有可能获得有限制的网络组件（例如路由器.防火墙.负载均衡器等）的控制。

2.系统业务与系统环境

2.1 业务情况

SaaS 物流业务在线上控制线下业务，主要考虑到线下业务周期长，反应慢，沟通效率不高，而当前网络技术发展迅速，网速慢早已不再是限制在线平台发展的主要问题，实时监控线下业务流程也变成了主流的复杂业务解决方案，该项目意在信息的流转进行全环节控制，服务于货代企业，通过提供接口等工具，帮助中小企业简单便利的对物流全环节沟通.控制等等。项目包括四个方面：

1. 权限管理：对登入系统的员工进行细粒度的权限控制
2. 货物管理：提供货物的全流程管理，包含商品详情，报价
3. 报运管理：包括购销合同，出口报运，装箱，委托，发票
4. 统计管理：以图形化界面的方式对销售，财务数据展示

2.2 开发环境与技术

后端框架采用 Spring +SpringMVC+mybatis +Dubbox，前端采用 AdminLTE 框架。Java 采用 JDK1.8，数据库使用 MySQL 5.7，maven 版本为

3.3.9。

2.2.1 SSM 创建流程

1. 创建 export_parent 父工程 (打包方式选择 pom)
2. 创建 export_web_manager 子模块 (打包方式是 war 包) -负责页面调用等逻辑
3. 创建 export_service_system 子模块 (打包方式是 jar 包) -负责业务逻辑
4. 创建 export_domain 子模块 (打包方式是 jar 包) -对应数据库表的 pojo
5. 创建 export_dao 子模块 (打包方式是 jar 包) -负责数据库连接方面的内容
6. 创建 export_common 子模块 (打包方式是 jar 包) -工具包
7. 在 export_parent 的 pom.xml 文件中引入坐标依赖

2.2.2 数据库环境

- 多租户技术 (Multi-TenancyTechnology) 又称多重租赁技术：是一种软件架构技术，是实现如何在多用户环境下 (此处的多用户一般是面向企业用户) 共用相同的系统或程序组件，并且可确保各用户间数据的隔离性。简单讲：在一台服务器上运行单个应用实例，它为多个租户 (客户) 提供服务。

从定义中我们可以理解：多租户是一种架构，目的是为了让多用户环境下使用同一套程序，且保证用户间数据隔离。那么重点就很浅显易懂了，多租户的重点就是同一套程序下实现多用户数据的隔离。

- 独立数据库：每个租户一个数据库。

优点：为不同的租户提供独立的数据库，有助于简化数据模型的扩展设计，满足不同租户的独特需求；如果出现故障，恢复数据比较简单。

缺点：增多了数据库的安装数量，随之带来维护成本和购置成本的增加。这种方案与传统的一个客户、一套数据、一套部署类似，差别只在于软件统一部署在运营商那里。

由此可见此方案用户数据隔离级别最高，安全性最好，但是成本较高

- 共享数据库、独立 Schema：即多个或所有的租户使用同一个数据库服务（如常见的 ORACLE 或 MySQL 数据库），但是每个租户一个 Schema。

优点：为安全性要求较高的租户提供了一定程度的逻辑数据隔离，并不是完全隔离；每个数据库可支持更多的租户数量。

缺点：如果出现故障，数据恢复比较困难，因为恢复数据库将牵涉到其他租户的数据；如果需要跨租户统计数据，存在一定困难。

这种方案是独立数据库的变种。只需要安装一份数据库服务，通过不同的 Schema 对不同租户的数据进行隔离。由于数据库服务是共享的，所以成本相对低廉。

- 共享数据库、共享数据表：即租户共享同一个 Database，同一套数据库表（所有租户的数据都存放在一个数据库的同一套表中）。在表中增加租户 ID 等租户标志字段，表明该记录是属于哪个租户的。

优点：所有租户使用同一套数据库，所以成本低廉。

缺点：隔离级别最低，安全性最低，需要在设计开发时加大对安全的开发量，数据备份和恢复最困难。

这种方案和基于传统应用的数据库设计没有任何区别，但是由于所有租户使用相同的数据库表，所以需要做好对每个租户数据的隔离安全性处理，这就增加了系统设计和数据管理方面的复杂程度。

- 三范式：

1. 第一范式（1NF）：确保每一列的原子性（做到每列不可拆分）

2. 第二范式（2NF）：在第一范式的基础上，非主字段必须依赖于主字段（一个表只做一件事）

3. 第三范式（3NF）：在第二范式的基础上，消除传递依赖

反三范式：

反三范式是基于第三范式所调整的，没有冗余的数据库未必是最好的数据库，有时为了提高运行效率，就必须降低范式标准，适当保留冗余数据。

2.3 分页查询业务

2.3.1 传统分页

我们之前分页是使用一个 PageBean 对象封装分页数据的。

对象中要定义 currentPage, pageSize, count, totalPage, List<T> 等属性，并在 dao 中编写相应的代码查询出分页结果封装到 PageBean 中。

2.3.2 PageHelper

PageHelper 是国内非常优秀的一款开源的 mybatis 分页插件，它支持基本主流与常用的数据库，例如 MySQL、oracle、mariaDB、DB2、SQLite、Hsqldb 等。

- PageHelper.startPage 静态方法调用：在需要进行分页的 MyBatis 查询方法前调用 PageHelper.startPage 静态方法即可，紧跟在这个方法后的第一个

MyBatis 查询方法会被进行分页。

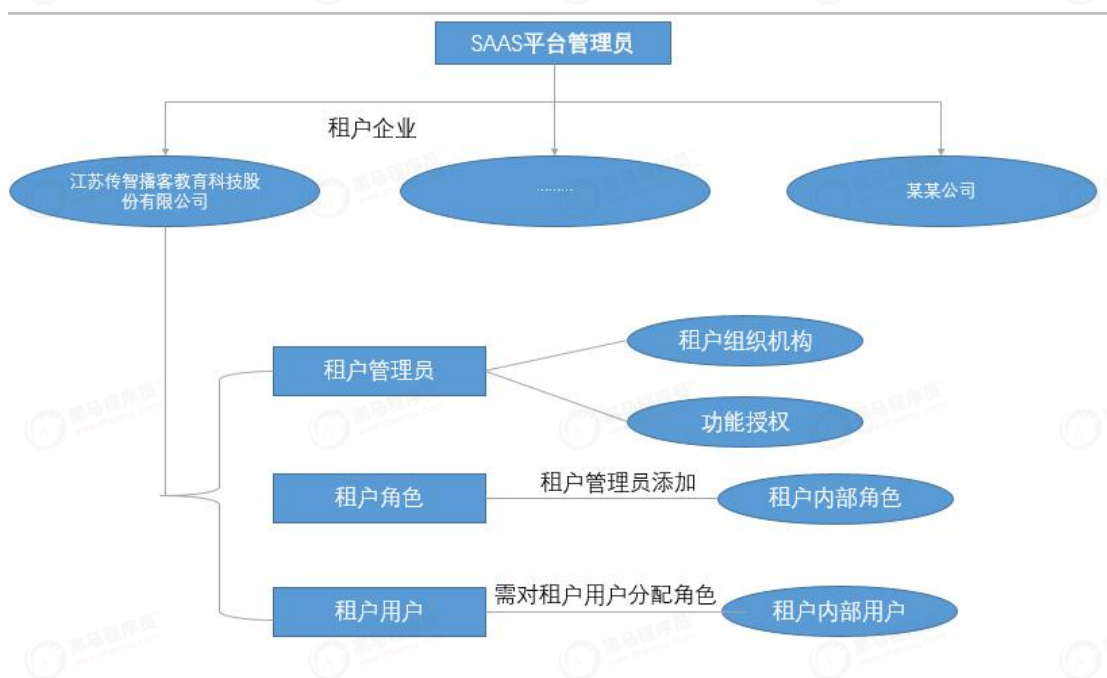
例：//获取第 1 页，10 条内容，默认查询总数 count

```
PageHelper.startPage(1, 10);
```

//紧跟着的第一个 select 方法会被分页

```
List<Country> list = countryMapper.selectIf(1);
```

2.4 权限



- SAAS 平台管理员：负责平台的日常维护和管理，包括用户日志的管理、租户账号审核、租户状态管理、租户费用的管理，要注意的是平台管理员不能对租户的具体业务进行管理。

- 企业租户：指访问 SaaS 平台的用户企业，在 SaaS 平台中各租户之间信息是独立的。

- 租户管理员：为租户角色分配权限和相关系统管理、维护。

- 租户用户：需对租户用户进行角色分配，租户用户只能访问授权的

模块信息。

2.5 动态构造菜单



1. 根据登录用户的用户信息查询模块。
2. SaaS 管理员，租户企业管理员直接查询即可。
3. 普通用户，通过用户的角色以及角色和模块之间的关联关系查询。
4. 将模块信息保存到 session 中。
5. 页面动态拼接菜单。

2.6 日志 (AOP)

AOP：面向切面编程。AOP 出现是 OOP 的延伸，是 OOP 的扩展。解决 OOP 中的一些问题。AOP 取代传统的纵向继承体系，采用横向抽取机制。

2.6.1 底层原理

Spring 实现 AOP 使用两种代理机制：

- * JDK 动态代理：基于接口实现
- * Cglib 动态代理：基于子类实现

2.6.2 注解介绍

@Aspect : 定义切面类

@Before :前置通知

@AfterReturning:后置通知

@Around :环绕通知

@AfterThrowing:异常抛出通知

@After :最终通知

@Pointcut :定义切入点

2.6.3 切面类.测试类举例

@Component

@Aspect

```
public class MyAspect {
```

```
    @Before("execution(*
```

```
cn.itcast.service.CustomerServiceImpl.add*(..))")
```

```
    public void before(){
```

```
        System.out.println("前置增强执行了。。。");
```

```
    }
```

```
}
```

@Test

```
public void test1(){
```

```
    ApplicationContext ac =new
```

```
    ClassPathXmlApplicationContext("applicationContext.xml");
```

```
CustomerService cs = (CustomerService) ac.getBean("customerService");

cs.updateCustomer();

}
```

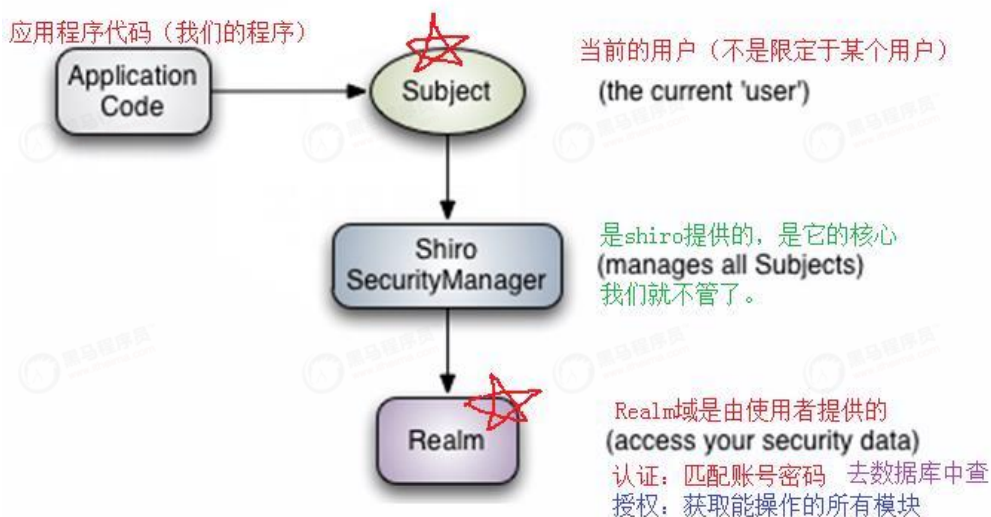
2.7 Shiro

2.7.1 传统登录方式

步骤如下：

- 1.Controller 接收请求数据 包含用户名和密码
- 2.调用 Service 去数据库查询 根据 Contoller 提供的用户名和密码
- 3.判断是否有结果返回
- 4.如果有返回对象 把返回的用户对象存入 Session 域中
- 5.如果没有返回对象 直接前往登录页面，并提示登录失败

2.7.2 Shiro 登录方式



2.7.3 使用步骤

第一步：在 maven 父工程中的 pom.xml 中导入坐标；

第二步：配置 spring 中提供的 shiro 过滤器（一当十的过滤器）；

第三步：在 spring 的配置文件中配置代理方式；

第四步：创建 shiro 的 spring 配置文件；

第五步：自定义 realm 域；

第六步：自定义密码比较器；

第七步：页面使用 shiro 标签，/home/title.jsp 主菜单。

2.7.4 Shiro 的缓存

为了提高认证和授权访问的效率，加入 Shiro 自带的 Ehcache。Shiro 每次授权都会通过 realm 获取权限信息，为了提高访问速度需要添加缓存，第一次从 realm 中读取权限数据，之后不再读取，这里 Shiro 和 Ehcache 整合。引入 Shiro 的 core 和 ehcache Jar 包。在 Shiro 的 xml 配置文件里边引入 缓存的配置。我们在项目中配置的是 shiro-ehcache.xml。里边主要配置了一些固定参数，还配置了缓存持久化的目录(序列化地址)。如果配置了缓存，那么在用户修改权限后，要注意清除缓存。

2.7.5 总结

学习 Shiro 框架我们就关心 3 点

1. Realm 域如何提供

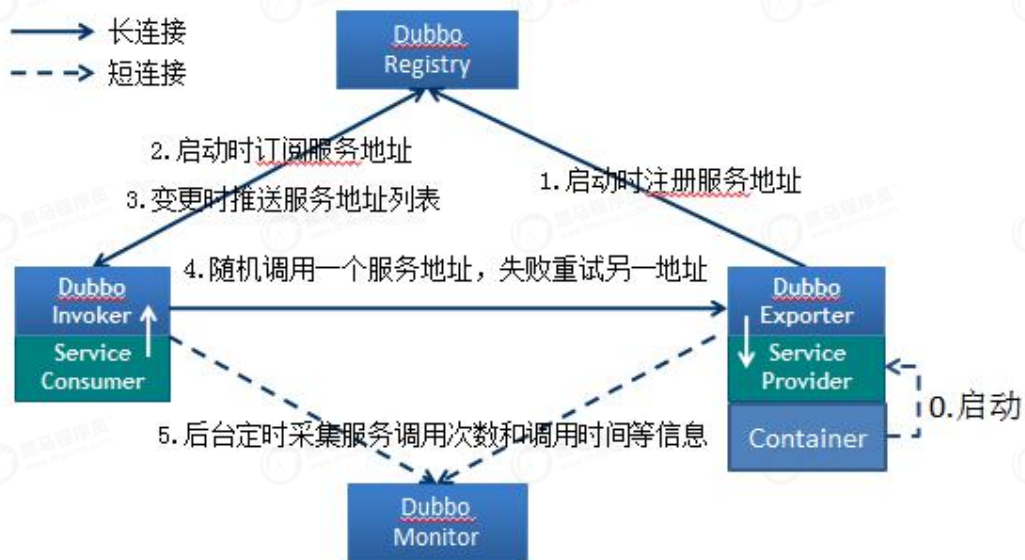
2. 密码比较器如何编写

3. Action 中的登录方法如何写

2.8 Dubbo

我们的项目采用了 SOA 的架构，利用 Dubbo+Zookeeper 的方式进行不同服务之间的注册以及调用，首先我跟大家介绍一下 Dubbo，Dubbo 致力于提供高性能和透明化的 RPC 远程服务调用方案，以及 SOA 服务治理方案。简单的说，dubbo 就是个服务框架，如果没有分布式的需求，其实是不需要用的，只有在分布式的时候，才有 dubbo 这样的分布式服务框架的需求，并且本质上是个服务调用的东东，说白了就是个远程服务调用的分布式框架。

2.8.1 Dubbo 的实现原理图如下



2.8.2 节点角色说明

Provider: 暴露服务的服务提供方。

Consumer: 调用远程服务的服务消费方。

Registry: 服务注册与发现的注册中心。

Monitor: 统计服务的调用次数和调用时间的监控中心。

Container: 服务运行容器。

2.8.3 调用关系说明

1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

2.8.4 在实际开发的场景中应该如何选择 RPC 框架

1. **SpringCloud**：Spring 全家桶，用起来很舒服，只有你想不到，没有它做不到。可惜因为发布的比较晚，国内还没出现比较成功的案例，大部分都是试水，不过毕竟有 Spring 作背景，还是比较看好。
2. **Dubbox**：相对于 Dubbo 支持了 REST，估计是很多公司选择 Dubbox 的一个重要原因之一，但如果使用 Dubbo 的 RPC 调用方式，服务间仍然存在 API 强依赖，各有利弊，懂的取舍吧。
3. **Thrift**：如果你比较高冷，完全可以基于 Thrift 自己搞一套抽象的自定义框架吧。
4. **Hessian**：如果是初创公司或系统数量还没有超过 5 个，推荐选择这个，毕竟在开发速度.运维成本.上手难度等都是比较轻量.简单的，即使在以后迁移至 SOA，也是无缝迁移。

5. **rpcx/gRPC** : 在服务没有出现严重性能的问题下, 或技术栈没有变更的情况下, 可能一直不会引入, 即使引入也只是小部分模块优化使用。

2.9 Zookeeper

ZooKeeper 是一种为分布式应用所设计的高可用.高性能且一致的开源协调服务, 它提供了一项基本服务: **分布式锁服务**。由于 ZooKeeper 的开源特性, 后来我们的开发者在分布式锁的基础上, 摸索出了其他的使用方法: **配置维护. 组服务. 分布式消息队列. 分布式通知/协调**等。

在我们的项目中 Zookeeper 作为服务注册中心, 他的主要任务是负责地址的注册以及查找, 相当于一个服务目录, 服务的提供者与服务的消费者只在启动时与注册中心进行交互, 注册中心也不会转发请求, 所以相对而言压力较小。

Zookeeper 诞生的背景:

其实除了 Zookeeper 实现了分布式锁以外还有 Google 的 Chubby 也是分布式锁的实现者, Chubby 是 Google 的闭源程序, 所以我们用不了, 而 Zookeeper 是雅虎模仿 Chubby 开发出来并捐献给 Apache 的, 所以才可以使用到这么优秀的开源程序去构建我们的分布式系统。

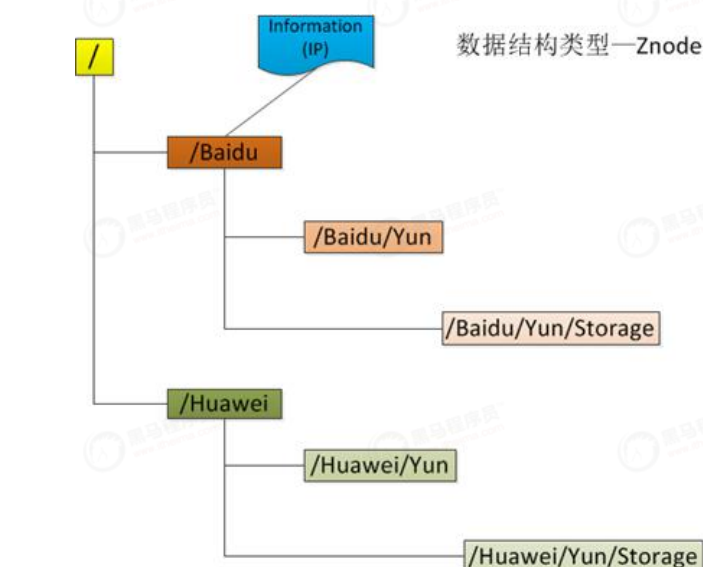
2.9.1 Zookeeper 是如何实现配置维护.组服务.分布式消息队列等等服务的呢?

首先 Zookeeper 在实现这些服务之前, 他设计了一种新的数据结构-Znode, 然后在这个数据结构的基础之上定义了一些对数据结构操作的方法, 另外 Zookeeper 还设计了一种通知机制-Watcher 机制。通过新的数据结构-Znode + 操作 Znode 的方法 + Watcher 机制实现了上述我们所看到的服务。

2.9.2 简述一下你对 Znode 的理解

首先跟你说一下 Znode 的结构跟我们日常中的 Window 环境中的文件目录结构很像，

下面放张截图



是不是很相似，Zookeeper 中的每一个节点（`/Baidu`）被称为 Znode。

虽然他们很相似，但是他们也有**不同之处**。

(1) 引用方式：

Znode 通过路径进行引用，就像 Linux 中的文件路径。并且路径都是绝对路径，因此他们都必须用“/”来开头，除此以外他们都必须唯一的，也就是每一个路径都只有一种表示，并且这些路径不可以改变。在 Zookeeper 中，这些路径是有 Unicode 字符串

组成,加入一些限制,比如:字符串“/zookeeper”就用以保存管理信息(关键配额信息等)

(2) Znode 结构

Znode 结构兼具了文件和目录两种特点,既可以保存数据.维护数据,又可以向目录一样作为路径表示的一部分。Znode 主要是由 3 个部分组成:

- ① Stat:这个表示状态信息,用于描述这个 Znode 的版本.权限等信息
- ② Data:这个表示与当前 Znode 相关联的数据
- ③ Children:这个表示当前 Znode 下的子节点

ZooKeeper 虽然可以关联一些数据,但并没有被设计为常规的数据库或者大数据存储,相反的是,它用来管理调度数据,比如分布式应用中的配置文件信息.状态信息.汇集位置等等。这些数据的共同特性就是它们都是很小的数据,通常以 KB 为大小单位。ZooKeeper 的服务器和客户端都被设计为严格检查并限制每个 Znode 的数据大小至多 1M,但常规使用中应该远小于此值。

(3) 数据访问

ZooKeeper 中的每个节点存储的数据要被原子性的操作。也就是说读操作将获取与节点相关的所有数据,写操作也将替换掉节点的所有数据。另外,每一个节点都拥有自己的 **ACL(访问控制列表)**,这个列表规定了用户的权限,即限定了特定用户对目标节点可以执行的操作。

(4) 节点类型

ZooKeeper 中的节点有两种,分别为临时节点和永久节点。节点的类型在创建时即被确定,并且不能改变。

- ① 临时节点:临时节点的生命周期依赖于创建它们的会话。一旦会话(Session)结束,临时节点将被自动删除,当然也可以手动删除。虽然每个临时的 Znode 都会绑定到一个客户端会话,但他们对所有的客户端还是可见的。另外,

ZooKeeper 的临时节点不允许拥有子节点。

② 永久节点 : 永久节点的生命周期不依赖于会话 , 并且只有在客户端显示执行删除操作的时候 , 他们才能被删除。

(5) 顺序节点

当创建 Znode 的时候 , 用户可以请求在 ZooKeeper 的路径结尾添加一个递增的计数。这个计数对于此节点的父节点来说是唯一的 , 它的格式为 "%10d"(10 位数字 , 没有数值的数位用 0 补充 , 例如 "0000000001")。当计数值大于 $2^{32}-1$ 时 , 计数器将溢出。

(6) 观察

客户端可以在节点上设置 watch , 我们称之为监视器。当节点状态发生改变时 (Znode 的增.删.改) 将会触发 watch 所对应的操作。当 watch 被触发时 , ZooKeeper 将会向客户端发送且仅发送一条通知 , 因为 watch 只能被触发一次 , 这样可以减少网络流量

2.9.3 Zookeeper 角色说明

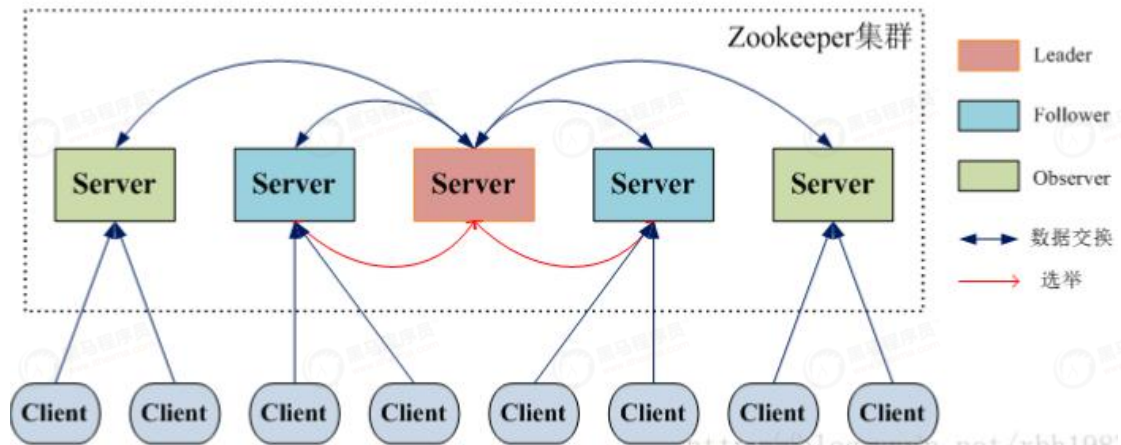
领导者 (leader) , 负责进行投票的发起和决议 , 更新系统状态

学习者 (learner) , 包括跟随者 (follower) 和观察者 (observer) , follower 用于接受客户端请求并想客户端返回结果 , 在选主过程中参与投票

Observer 可以接受客户端连接 将写请求转发给 leader , 但 observer 不参加投票过程 , 只同步 leader 的状态 , observer 的目的是为了扩展系统 , 提高读取速度

客户端 (client) , 请求发起方

2.9.4 角色调用关系说明



2.9.5 Zookeeper 的核心是什么？

Zookeeper 的核心是原子广播，这个机制保证了各个 Server 之间的同步。实现这个机制的协议叫做 Zab 协议。Zab 协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式，当领导者被选举出来，且大多数 Server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 Server 具有相同的系统状态。

2.9.6 Zookeeper 中的每个 Server 有几种状态？

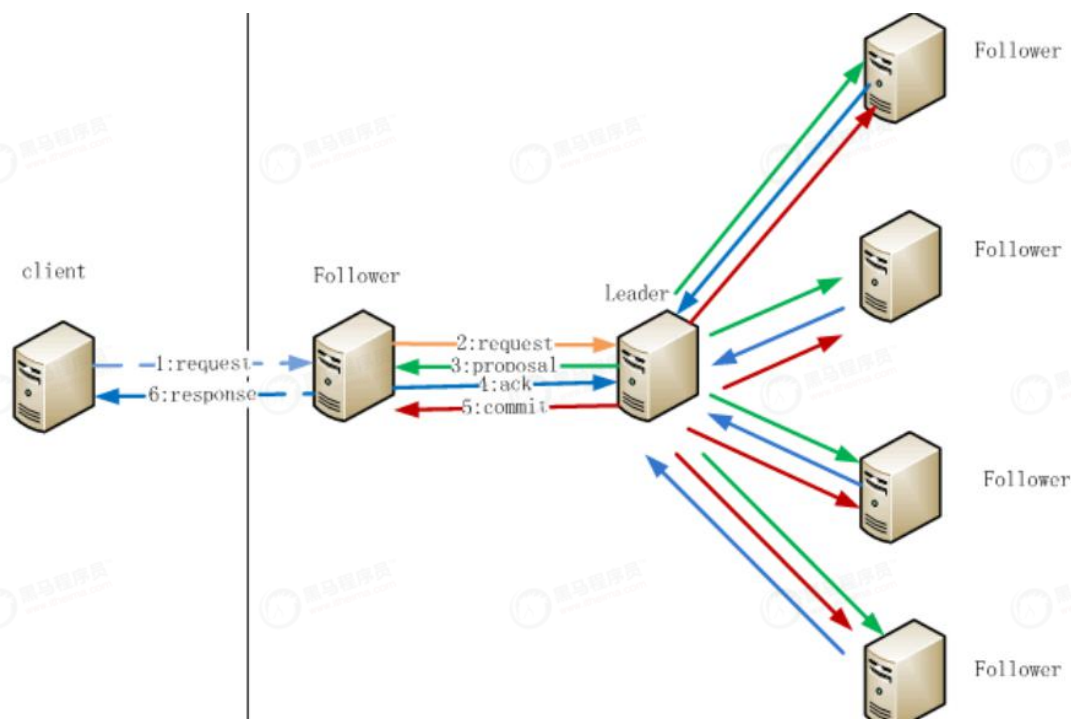
每个 Server 在工作过程中有三种状态：

LOOKING：当前 Server 不知道 leader 是谁，正在搜寻

LEADING：当前 Server 即为选举出来的 leader

FOLLOWING：leader 已经选举出来，当前 Server 与之同步

2.9.7 Zookeeper 的节点数据操作流程



1. 在 Client 向 Follower 发出一个写的请求
2. Follower 把请求发送给 Leader
3. Leader 接收到以后开始发起投票并通知 Follower 进行投票
4. Follower 把投票结果发送给 Leader
5. Leader 将结果汇总后如果需要写入，则开始写入同时把写入操作通知给 Leader，然后 commit;
6. Follower 把请求结果返回给 Client

2.9.8 为什么 zookeeper 集群的数目，一般为奇数个？

1.容错

由于在增删改操作中需要半数以上服务器通过，来分析以下情况。

2 台服务器 , 至少 2 台正常运行才行 (2 的半数为 1 , 半数以上最少为 2) ,
正常运行 1 台服务器都不允许挂掉

3 台服务器 , 至少 2 台正常运行才行 (3 的半数为 1.5 , 半数以上最少为 2) ,
正常运行可以允许 1 台服务器挂掉

4 台服务器 , 至少 3 台正常运行才行 (4 的半数为 2 , 半数以上最少为 3) ,
正常运行可以允许 1 台服务器挂掉

5 台服务器 , 至少 3 台正常运行才行 (5 的半数为 2.5 , 半数以上最少为 3) ,
正常运行可以允许 2 台服务器挂掉

6 台服务器 , 至少 3 台正常运行才行 (6 的半数为 3 , 半数以上最少为 4) ,
正常运行可以允许 2 台服务器挂掉

通过以上可以发现 , 3 台服务器和 4 台服务器都最多允许 1 台服务器挂掉 ,
5 台服务器和 6 台服务器都最多允许 2 台服务器挂掉

但是明显 4 台服务器成本高于 3 台服务器成本 , 6 台服务器成本高于 5 服务器成本。这是由于半数以上投票通过决定的。

2.防脑裂

一个 zookeeper 集群中 , 可以有多个 follower.observer 服务器 , 但是必需只能有一个 leader 服务器。

如果 leader 服务器挂掉了 , 剩下的服务器集群会通过半数以上投票选出一个新的 leader 服务器。

集群互不通讯情况 :

一个集群 3 台服务器 , 全部运行正常 , 但是其中 1 台裂开了 , 和另外 2 台无法通讯。3 台机器里面 2 台正常运行过半票可以选出一个 leader。

一个集群 4 台服务器，全部运行正常，但是其中 2 台裂开了，和另外 2 台无法通讯。4 台机器里面 2 台正常工作没有过半票以上达到 3，无法选出 leader 正常运行。

一个集群 5 台服务器，全部运行正常，但是其中 2 台裂开了，和另外 3 台无法通讯。5 台机器里面 3 台正常运行过半票可以选出一个 leader。

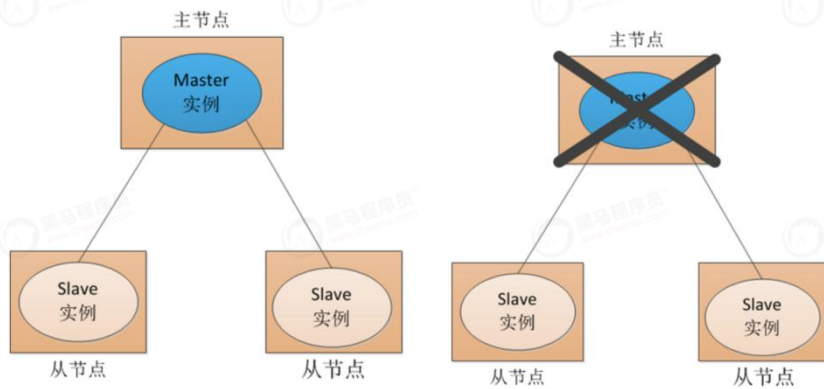
一个集群 6 台服务器，全部运行正常，但是其中 3 台裂开了，和另外 3 台无法通讯。6 台机器里面 3 台正常工作没有过半票以上达到 4，无法选出 leader 正常运行。

通过以上分析可以看出，为什么 zookeeper 集群数量总是单出现，主要原因还是在于第 2 点，防脑裂，对于第 1 点，无非是正本控制，但是不影响集群正常运行。但是出现第 2 种裂的情况，zookeeper 集群就无法正常运行了。

2.9.9 经典实例：Zookeeper 分布式锁的应用场景

背景：

在分布式锁服务中，有一种最典型应用场景，就是**通过对集群进行 Master 选举，来解决**分布式系统中的**单点故障**。什么是分布式系统中的单点故障：通常分布式系统采用主从模式，就是一个主控机连接多个处理节点。主节点负责分发任务，从节点负责处理任务，当我们的主节点发生故障时，那么整个系统就都瘫痪了，那么我们把这种故障叫作单点故障。如下图 1.1 和 1.2 所示：

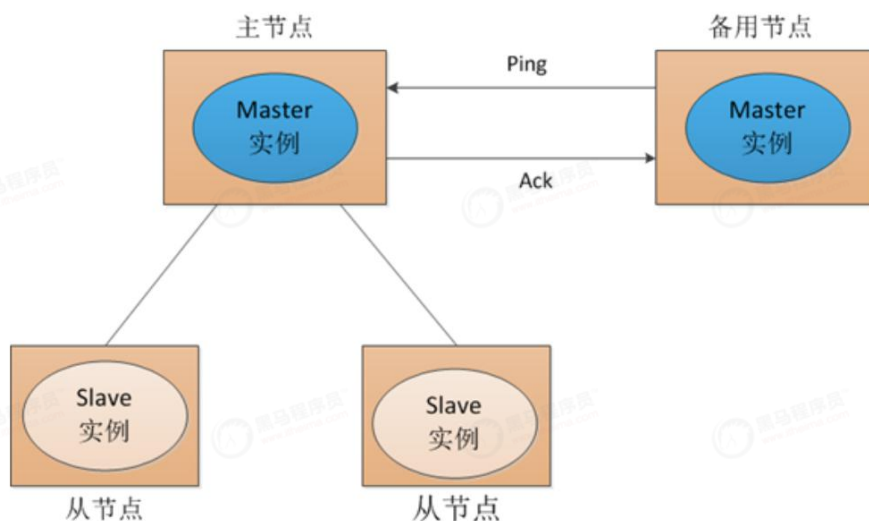


1.1 主从模式分布式系统

1.2 单点故障

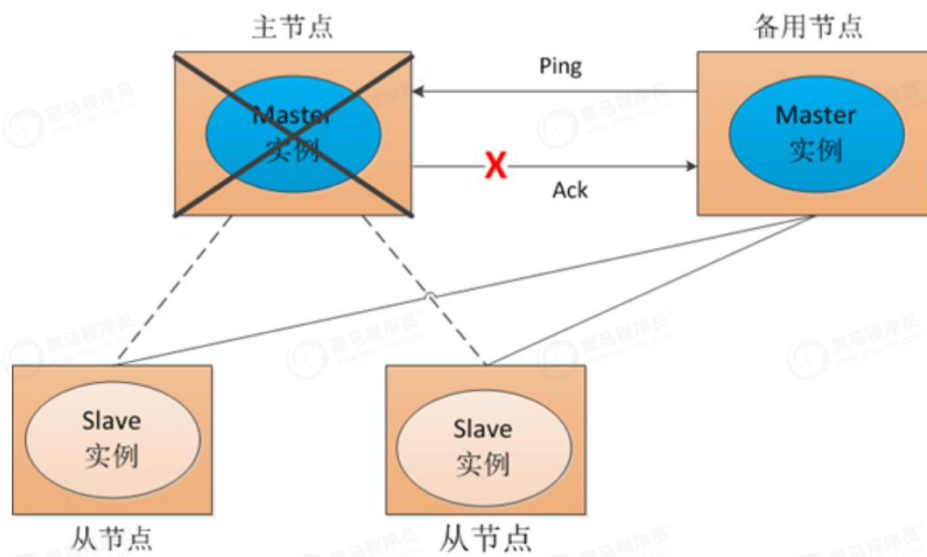
传统解决方案

传统方式是采用一个**备用节点**，这个备用节点**定期给当前主节点发送 ping 包**，**主节点收到 ping 包以后向备用节点发送回复 Ack**，当备用节点收到回复的时候就会认为当前主节点还活着，让他继续提供服务。如图 1.3 所示



1.3 传统解决方案

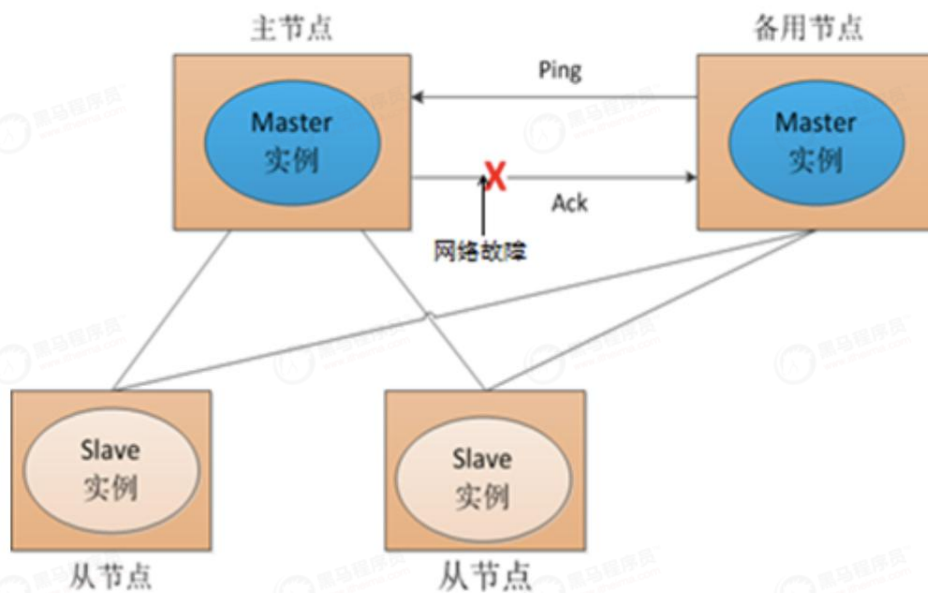
当主节点挂了，这时候备用节点收不到回复了，然后他就认为主节点挂了接替他



成为主节点如下图 1.4 所示

1.4 传统解决方案

但是这种方式就是有一个隐患，就是网络问题，来看一网络问题会造成什么后果，如下图 1.5 所示



1.5 网络故障

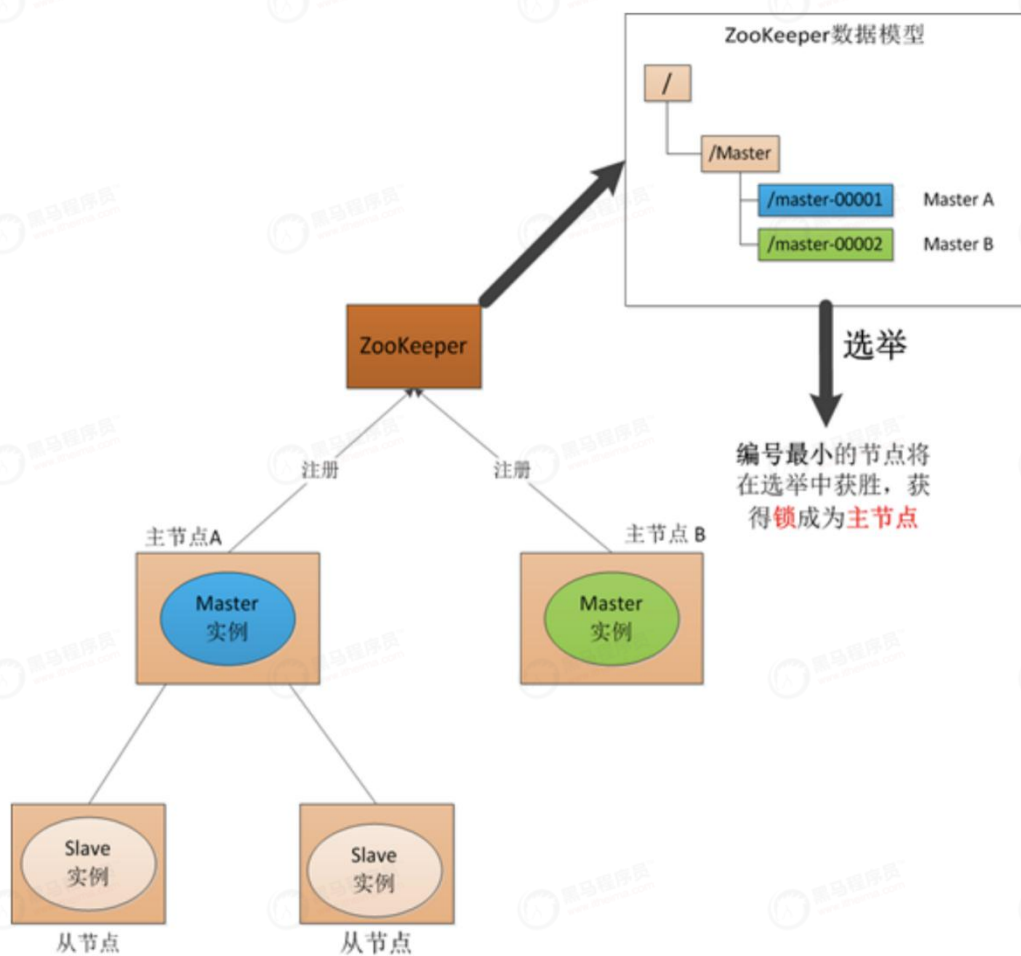
传统解决方案总结：

也就是说我们的主节点的并没有挂，只是在回复的时候网络发生故障，这样我们的备用节点同样收不到回复，就会认为主节点挂了，然后备用节点将他的 Master 实例启动起来，这样我们的分布式系统当中就有了两个主节点也就是-双 Master，出现 Master 以后我们的从节点就会将它所做的事一部分汇报给了主节点，一部分汇报给了从节点，这样服务就全乱了。

Zookeeper 解决方案：

Master 启动：

在引入了 Zookeeper 以后我们启动了两个主节点，"主节点-A"和"主节点-B"他们启动以后，都向 ZooKeeper 去注册一个节点。我们假设"主节点-A"所注册地节点是"master-00001"，"主节点-B"所注册的节点是"master-00002"，注册完以后进行选举，**编号最小的节点将在选举中获胜获得“锁”并且成为主节点**，也就是我们的"主节点-A"将会获得“锁”成为主节点，然后"主节点-B"将被**阻塞**成为一个**备用节点**。那么，通过这种方式就完成了对两个 Master 进程的调度。如图 2.1 所示

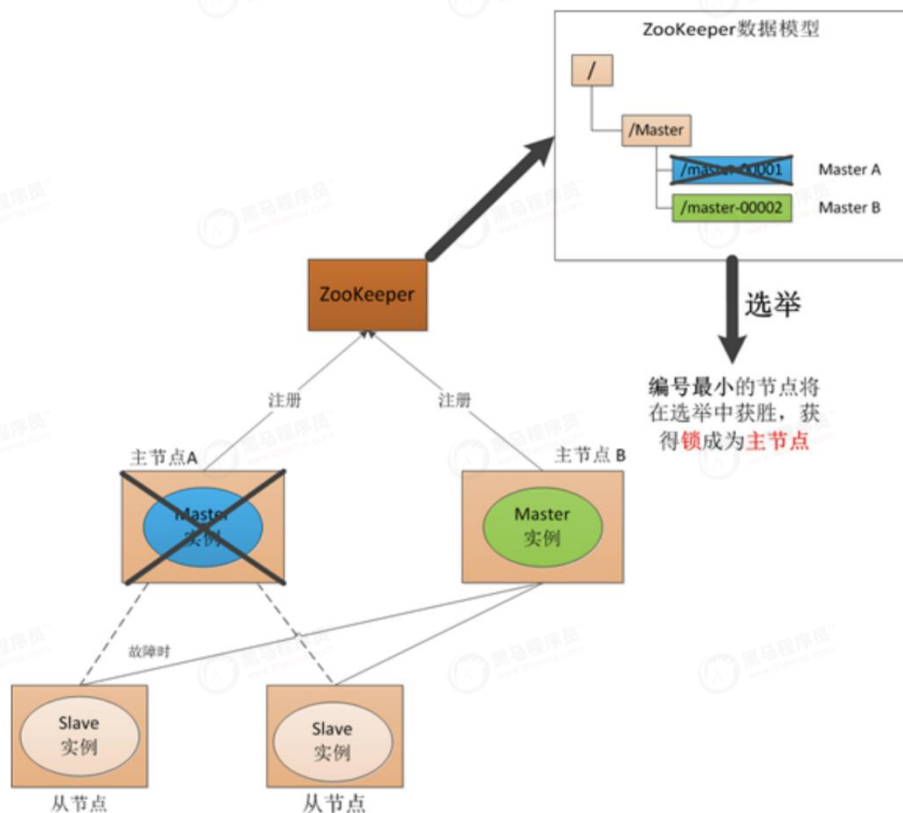


2.1 Zookeeper Master 选举

(1) Master 故障

如果“主节点-A”挂了，这时候他所注册的节点将被自动删除，

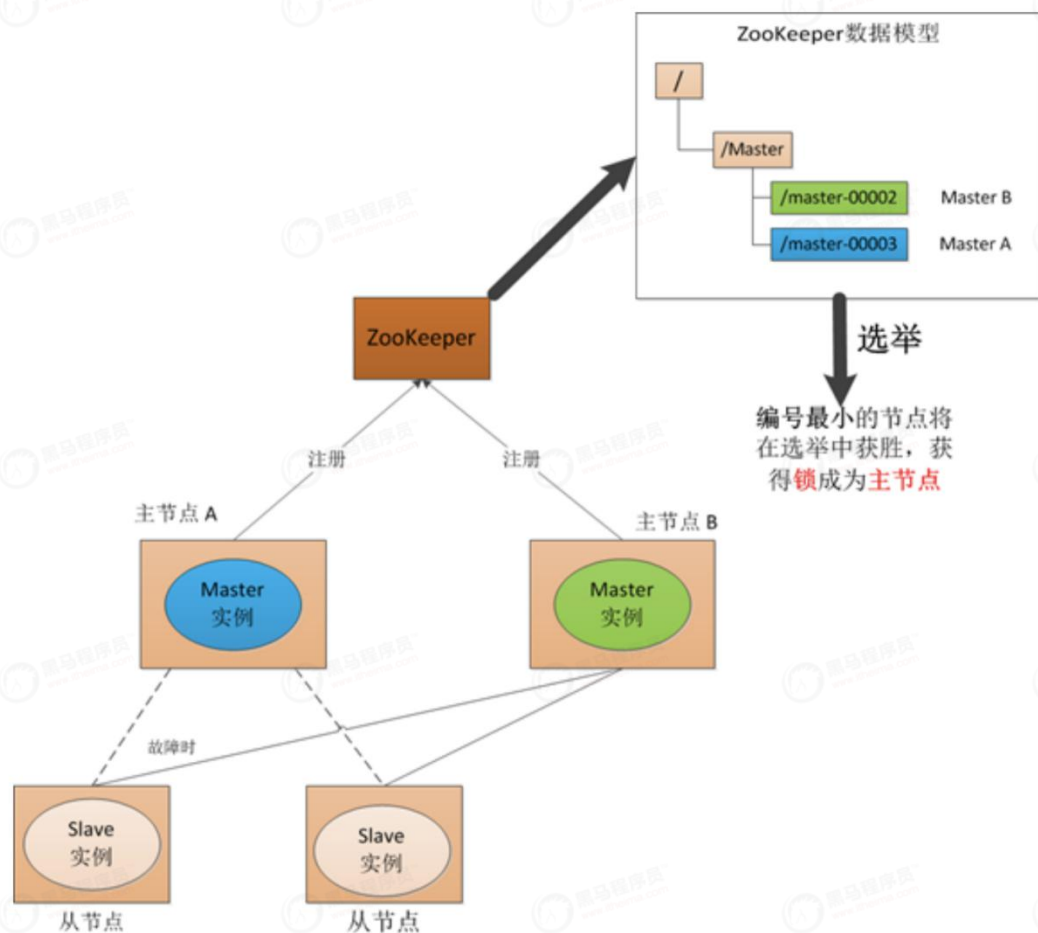
ZooKeeper 会自动感知节点的变化，然后再次发出选举，这时候“主节点-B”将在选举中获胜，替代“主节点-A”成为主节点。如图 2.2 所示



2.2 Master 故障

(2) Master 恢复

如果“主节点-A”恢复了，他会再次向 ZooKeeper 注册一个节点，这时候他注册的节点将会是“master-00003”，ZooKeeper 会感知节点的变化再次发动选举，由于编号最小的节点将在选举中获胜获得“锁”并且成为主节点，这时候“主节点-B”在选举中会再次获胜继续担任“主节点”，“主节点-A”会担任备用节点。如图 2.3 所示：



2.3 Master 恢复

2.10 Mybatis 的逆向工程

2.10.1 Mybatis 逆向工程概述

Mybatis 是目前很流行的持久层框架，很多企业都在采用。但是其复杂繁琐的配置，重复性的实体类创建等等，消耗了程序员大量的精力，同时有些地方如果一个细小的疏忽，可能导致最终功能运行失败。例如：在几十个字段的表中，某一列的列名配置疏忽。基于此，Mybatis 推出了一套 jar 包，可以依据我们设计好的数据库表，自动生成 pojo.mapper 以及 mapper.xml。有了逆向工程，便大大缩减了我们的开发时间。本章节将介绍借助 idea 的方式实现 Mybatis 的逆向工程。

2.11 POI 技术

POI 是用 JAVA 语言编写的免费的开源的跨平台 API，可以对微软的 word.Excel.PPT 进行操作。

2.11.1 Excel 报表导出流程

1. 创建工作簿(HSSFWorkbook)<工作簿这块我们需要注意一个问题，普通处理 Excel 我们分为 Excel03(HSSF)版和 07(XSSF)版本以上，对于大数据处理我们使用 SXSSF，比如一个 Excel 有 40 多列>

2. 创建工作表(createSheet())

3. 设置一些参数(sheet.setColumnWidth(0, 2*256);)设置列宽，我们一般都会*256，因为在 HSSFWorkbook 的底层源码里边对列设置的方法里边，做的处理是二百五十六分之一(1/256)，所以我们要设置列宽的时候，乘以 256 就是我们想要的值。

4. 根据项目需求我们可能要需要设置标题，设置行高(setHeightInPoints())，设置单元格对象，设置对象里边的内容。还有就是合并单元格，横向合并单元格(sheet.addMergedRegion)等。

5. 把在后台查询的数据填充到对应 Excel 表格里边。

6. 实现文件下载，导出到制定路径。

7. 根据客户提供的模板和要求做样式的调整，比如：大标题，小标题，字体，居中设置等。

因为不同企业对导出的表格要求不一致，为了便于快速开发，我们在公司一般采取的是模板打印。具体的实现思路是：在后台查询要导入的数据，**制作模版**

文件（模版文件的路径） 导入（加载）模版文件，从而得到一个工作簿 读取工

作表 读取行 读取单元格 读取单元格样式 设置单元格内容 其他单元格就可以使用读到的样式了 。

针对于 Excel 的百万级别报表数据导出，我们使用 POI 提供的 SXSSFWork 处理大数据量。**但是他不支持模板打印。**

2.11.2 百万级数据导出

针对于大宗贸易我们，涉及的数据量比较大，我们这里采用 POI 给我们提供的支持 07 版本的 SXSSFWork 对象实现导入导出，但是这个也有一个不好的地方就是不能使用模板打印，不能使用太多的样式。需要在后台根据客户需求进行设置。为了比较 POI 里边的 XSSFwork 和 SXSSFwork 我们这里可以借助工具 Jvisualvm 进行测评在实现大数据量导出的时候，进行系统监控，比如：CPU 利用率，gc 监控，堆利用率，内存利用率，类监控，线程监控。

但是在使用 SXSSFWork 进行导入的时候，因为数据量比较大，经常会造成内存溢出，我们这里使用 POI 基于事件模式解析案例提供的 Excel 文件。**其实 POI 里边提供了两种模式，用户模式(系统默认)和事件模式。这两个的区别是：用户模式一次性加载所有内容到内存，然后着个解析。事件模式是一边加载一边解析。**

使用这个进行导入时，我们需要定义两个解析器，一个是 Excel 解析器，这个公司直接封装好的，拿过来直接用。还有一个是 sheet 的解析器。Sheet 解析器是基于 SAX 解析(SAX 提供了一种从 XML 文档中读取数据的机制。它逐行扫描文档，一边扫描一边解析。由于应用程序 只是在读取数据时检查数据，因此不需要将数据存储在内中，这对于大型文档的解析是个巨大优势)进行实现的。我们需要在这个解析器里边定义开始行和结束行，创建解析行的方法，在方

法里边通过 `switch {case }` 语句根据要导入的信息确定值,我们这里用的每一列的表示符(比如 : A,B , C 等)进行数据处理。

原理分析 :借助临时存储空间生成 excel 在实例化 SXSSFWork 这个对象时,可以指定在内存中所产生的 POI 导出相关对象的数量(默认 100),一旦内存中的对象的个数达到这个指定值时,就将内存中的这些对象的内容写入到磁盘(XML 的文件格式),就可以 将这些对象从内存中销毁,以后只要达到这个值,就会以类似的处理方式处理,直至 Excel 导出完成。

2.12 WebService 技术

webService 是这项技术指的是在自己网站调用其他网站或者程序。

webService 指的是一套规范或者服务,主要是为了便于我们跨应用,跨网站进行调用。它约定了不同程序之间的规范,基于 xml 形式进行数据通信。我们在项目开发中使用的是基于 webService 的一套框架 CXF, Cxf 是基于 SOA 总线结构,依靠 **spring 完成模块的集成**,实现 SOA 方式。灵活的部署:可以运行在 Tomcat,Jboss,Jetty(内置),weblogic 上面。

在工作中,经常会遇到不同公司系统之间的远程服务调用。远程调用技术非常多,如 rmi.netty.mina.hessian.dubbo.Motan.SpringCloud.webservice 等等。但是我个人还是比较喜欢 webService 的,主要原因有 4 点:

1.webservice 技术是建立在 http+xml 基础之上的,非常的轻量级。

2.webservice 技术可通过 wsdl 来定义调用关系,双方系统可根据 wsdl 快速的进行开发对接。

3.webservice 是一种标准,有各种语言对它的实现,支持异构系统之间的对接。

4. 必要情况下，还可以使用 httpclient 作为客户端进行调用，以降低依赖。

2.12.1 webService 的三大规范是什么？

分别指的是 JAX-WS，JAX-RS，JAX-WS&SAAJ(废弃)

JAX-RS 是一个 Java 编程语言接口，被设计用来简化使用 REST 架构的应用程序的开发。

借助 Jax-RS 规范提供一些注解和配置方式：快速的搭建一个符合 Restful 风格的应用。

借助 Jax-RS 规范提供的工具类，方便的调用符合 Jax-RS 的应用。

2.12.2 除了 webService，还有什么技术可以实现内部程序调用其他网站服务？

目前还有 HttpClient.Dubbo<主要应用于内部系统访问>.RMI.Socket。

HttpClient

HttpClient 是 Apache Jakarta Common 下的子项目，用来提供高效的、最新的、功能丰富的支持 HTTP 协议的客户端编程工具包，并且它支持 HTTP 协议最新的版本和建议。主要在调用一些对外的 API 接口时使用。

Web Service

Web Service 提供的服务是基于 web 容器的，底层使用 http 协议，类似一个远程的服务提供者，比如天气预报服务，对各地客户端提供天气预报，是一种请求应答的机制，是跨系统跨平台的。就是通过一个 servlet，提供服务出去。

首先客户端从服务器的到 WebService 的 WSDL，同时在客户端声称一个代理类(Proxy Class) 这个代理类负责与 WebService 服务器进行 Request 和 Response 当一个数据 (XML 格式的) 被封装成 SOAP 格式的数据流发送到服务器端的时候，就会生成一个进程对象并且把接收到这个 Request 的 SOAP 包

进行解析，然后对事物进行处理，处理结束以后再对这个计算结果进行 SOAP 包装，然后把这个包作为一个 Response 发送给客户端的代理类(Proxy Class)，同样地，这个代理类也对这个 SOAP 包进行解析处理，继而进行后续操作。这就是 WebService 的一个运行过程。

Web Service 大体上分为 5 个层次:

1. Http 传输信道
2. XML 的数据格式
3. SOAP 封装格式
4. WSDL 的描述方式 从下往上看这个说明书。
5. UDDI UDDI 是一种目录服务 ,企业可以使用它对 Webservices 进行注册和搜索

2.12.3 如何以 JAVA 的形式启动当前 web 应用

- 1.创建工厂
- 2.制定接口
- 3.制定交互对象
- 4.配置日志输出
- 5.启动服务

2.12.4 远程通信的几种选择 (RPC , Webservice , RMI , JMS 的区别)

RMI (Remote Method Invocation)

RMI 采用 stubs 和 skeletons 来进行远程对象(remote object)的通讯。stub 充当远程对象的客户端代理，有着和远程对象相同的远程接口，远程对象的调用实际是通过调用

该对象的客户端代理对象 stub 来完成的,通过该机制 RMI 就好比它是本地工作,采用 tcp/ip 协议,客户端直接调用服务端上的一些方法。优点是强类型,编译期可检查错误,缺点是只能基于 JAVA 语言,客户机与服务器紧耦合。

JMS (Java Messaging Service)

JMS 是 Java 的消息服务,JMS 的客户端之间可以通过 JMS 服务进行异步的消息传输。

JMS 支持两种消息模型: Point-to-Point (P2P) 和 Publish/Subscribe (Pub/Sub),即点对点 and 发布订阅模型。

2.12.5 webService 的三要素是什么?

SOAP: 基于 HTTP 协议,采用 XML 格式,用来传递信息的格式。

WSDL: 用来描述如何访问具体的服务。

UDDI: 用户自己可以按 UDDI 标准搭建 UDDI 服务器,用来管理,分发,查询 WebService 。其他用户可以自己注册发布 WebService 调用。

2.13 PDF 导出

2.13.1 常用的 pdf 技术有哪些?

1.iText PDF :iText 是著名的开放项目,是用于生成 PDF 文档的一个 java 类库。通过 iText 不仅可以生成 PDF 或 rtf 的文档,而且可以将 XML.Html 文件转化为 PDF 文件。

1. Openoffice : openoffice 是开源软件且能在 windows 和 linux 平台下运行,可以灵活的将 word 或者 Excel 转化为 PDF 文档。

2. **Jasper Report** : 是一个强大灵活的报表生成工具,能够展示丰富的页面内容,并将之转换成 PDF 。

我们开发中一般选用的是 Jasper Report 技术,这个技术完全由 Java 写

成，同时还有对应的工具 Jaspersoft Studio，在线编辑很方便。支持多种表格的输出，同时支持多种数据源，通过 JASPER 文件及数据源，JASPER 就能生成最终用户想要的文档格式。

2.13.2 项目中 PDF 导出的使用

我们在项目中先 1.导入坐标，2.引入模板，把编译好的模板引入到当前工程中。3.配置控制器方法(1.加载模板文件，2.构建文件输入流；3.创建 JasperPrint 对象；4.写入 pdf 文档输出)。导入导出的流程一般都是固定的格式，我自己写了个文档，当我使用的时候，我会拿出来按照上边的步骤进行操作完成。

1. 引入 jasper 文件

2. 构造数据

- a.报运单数据

- b.报运货物列表(通过 list 集合创建 javaBean 的数据源对象)

2.13.3 简述 JasperReport 的生命周期

设计 (Design) 阶段.执行 (Execution) 阶段以及输出 (Export) 阶段。

设计阶段就是创建模板，模板创建完成我们保存为 JRXML 文件 (JR 代表 JasperReports) ,其实就是一个 XML 文件。

执行阶段编译成可执行的二进制文件(即.Jasper 文件)结合数据进行执行，进行数据填充。

输出阶段 (Export): 数据填充结束，可以指定输出为多种形式的报表。

2.13.4 JasperReport 的执行流程是什么？

- JRXML:报表填充模板，本质是一个 XML.
- Jasper:由 JRXML 模板编译生成的二进制文件，用于代码填充数据。

- Jprint:当用数据填充完 Jasper 后生成的文件，用于输出报表。
- Exporter:决定要输出的报表为何种格式，报表输出的管理类。
- Jasperreport 可以输出多种格式的报表文件，常见的有

Html,PDF,xls 等

2.14 ActiveMQ

2.14.1 消息分发模式有几种？分别有什么特点？

点对点模式(P2P)和发布-订阅模式(Pub/Sub)。 点对点是每个消息都被发送到特定的消息队列，接收者从队列中获取消息。队列保留着消息，直到他们被消费或超时。**发布/订阅(Publish-Subscribe)** :包含三个角色：主体(Topic)，发布者(Publisher)，订阅者(Subscriber)，多个发布者将消息发送到 topic，系统将这些消息投递到订阅此 topic 的订阅者。

点对点模型的特点：

每个消息只有一个消费者(Consumer) (即一旦被消费，消息就不再在消息队列中)；

发送者和接收者之间在时间上没有依赖性，也就是说当发送者发送了消息之后，不管接收者有没有正在运行，它不会影响到消息被发送到队列；

接收者在成功接收消息之后需向队列应答成功；

发布/订阅模型的特点：

每个消息可以有多个消费者；

发布者和订阅者之间有时间上的依赖性。

针对某个主题(Topic)的订阅者，它必须创建一个订阅者之后，才能消费发布者的消息，而且为了消费消息，订阅者必须保持运行的状态；

2.14.2 结合项目聊一聊消息中间件使用场景

我们在项目中主要在新注册用户给用户发邮箱(在用户注册时，因为我们这边会给用户发送邮件，以确定用户是否添加。那么会遇到网络的原因造成用户迟迟收不到邮件，那么我们把发送的事情放到消息队列中。)，到货通知，催单通知。以及电商中的数据同步，比如：数据库和索引库进行同步；页面静态化；

2.14.3 activeMQ 收不到消息的原因及解决方式

时间长了就会出现，卡死，新的数据不能从队列接收到。只能重启程序。

解决方案：

1) 不要频繁的建立和关闭连接 JMS 使用长连接方式，一个程序，只要和 JMS 服务器保持一个连接就可以了，不要频繁的 建立和关闭连接。频繁的建立和关闭连接，对程序的性能影响还是很大的。这一点和 JDBC 还是不太一样的。

2) Connection 的 start()和 stop()方法代价很高 JMS 的 Connection 的 start()和 stop()方法代价很高，不能经常调用。我们试用的时候，写了个 jms 的 connection pool，每次将 connection 取出 pool 时调用 start()方法，归还时调用 stop() 方法，然而后来用 jprofiler 发现，一般的 cpu 时间都耗在了这两个方法上。

3) start()后才能收消息 Connection 的 start()方法调用后，才能收到 jms 消息。如果不调用这个方法，能发出消息，但是一直收不到消息。不知道其它的 jms 服务器也是这样。 4) 显式关闭 Session 如果忘记了最后关闭 Connection 或 Session 对象，都会导致内存泄漏。这个在我测试的时候也发现了。本来以为关闭了 Connection，由这个 Connection 生成的 Session 也会被自动关闭，结果并非如此，Session 并没有关闭，导致内存泄漏。所以一

定要显式的关闭 Connection 和 Session。

5) 对 Session 做对象池 对 Session 做对象池，而不是 Connection。

Session 也是昂贵的对象，每次使用都新建和关闭，代价也非常高。而且后来我们发现，原来 Connection 是线程安全的，而 Session 不是，所以 后来改成了对 Session 做对象池，而只保留一个 Connection。

2.14.4 activeMQ 存在发生消息太大，造成消息接受不成功

多个线程从 activeMQ 中取消息，随着业务的扩大，该机器占用的网络带宽越来越高。

仔细分析发现，mq 入队时并没有异常高的网络流量，仅仅在出队时会产生很高的网络流量。

最终发现是 spring 的 jmsTemplate 与 activemq 的 prefetch 机制配合导致的问题。

研究源码发现 jmsTemplate 实现机制是：每次调用 receive()时都会创建一个新的 consumer 对象，用完即销毁。

正常情况下仅仅会浪费重复创建 consumer 的资源代价，并不至于产生正常情况十倍百倍的网 络流量。

但是 activeMQ 有一个提高性能的机制 prefetch ,此时就会有严重的问题。

prefetch 机制：每次 consumer 连接至 MQ 时，MQ 预先存放许多 message 到消费者（前提是 MQ 中存在大 量消息），预先存放 message 的数量取决于 prefetchSize（默认为 1000）。此机制的目的很显然，是想让客户端代码用一个 consumer 反复进行 receive 操作，这样能够大量提高出队 性能。

此机制与 `jmsTemplate` 配合时就会产生严重的问题，每次 `jmsTemplate.receive()`，都会产生 1000 个消息的网络流量，但是因为 `jmsTemplate` 并不会重用 `consumer`，导致后面 999 个消息都被废弃。反复 `jmsTemplate.receive()` 时，表面上看不出任何问题，其实网络带宽会造成大量的浪费。

解决方案：

1.若坚持使用 `jmsTemplate`，需要设置 `prefetch` 值为 1，相当于禁用了 `activeMQ` 的 `prefetch` 机制，此时感觉最健壮，就算多线程，反复调用 `jmsTemplate.receive()` 也不会有任何问题。但是会有资源浪费，因为要反复创建 `consumer` 并频繁与服务器进行数据通信，但在性能要求不高的应用中也不算什么问题。

2.不使用 `jmsTemplate`，手工创建一个 `consumer`，并单线程反复使用它来 `receive()`，此时可以充分利用 `prefetch` 机制。配合多线程的方式每个线程拥有自己的一个 `consumer`，此时能够充分发挥 MQ 在大吞吐量时的速度优势。

切记避免多线程使用一个 `consumer` 造成的消息混乱。大吞吐量的应用推荐使用方案 2，能够充分利用 `prefetch` 机制提高系 MQ 的吞吐性能。

2.15 Quartz

1. 项目中的定时任务我们一般使用 Quartz 或者 `SpringTask` 进行实现。我们项目中使用的是 Quartz，是一个完全由 Java 编写的开源任务调度的框架，通过触发器设置作业定时运行规则，控制 作业的运行时间。其中 quartz 集群

通过故障切换和负载均衡的功能，能给调度器带来高可用性和伸缩性。主要用来执行定时任务，如：定时发送信息、定时生成报表等等。

2. Quartz 在项目中的使用。

- (1) 搭建环境(引入 Quartz 坐标)
- (2) 配置一个类，配置定时任务执行的方法
- (3) 将此类交给 spring 进行管理
- (4) 配置 Quartz 和 spring 整合

我们在开发中经常为如何准确的定位好时间，我们这里可以使用 cron.qqe2.com 在线 cron 表达式生成。Cron 他是按照<秒，分钟，小时，日，月，周，年>进行算的。

```
<!--1.将定时任务类交给spring容器-->
<bean id="myTask" class="cn.itcast.web.task.MyTask"></bean>

<!--2.配置jobDetail： 配置定时执行的类和方法-->
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="myTask"></property>
    <property name="targetMethod" value="excete"></property>
</bean>

<!--3.配置trigger: (触发器) 配置时间以及jobdetail关系 -->
<bean id="tigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">
    <!--cron表达式-->
    <property name="cronExpression" value="0/5 * * * ? *"></property>
    <property name="jobDetail" ref="jobDetail"></property>
</bean>

<!--4.配置定时任务管理器-->
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="tigger"></ref>
        </list>
    </property>
</bean>
</beans>
```

Quartz 和 SpringTask 的区别是什么？(参考链接：

<https://blog.csdn.net/swl979623074/article/details/79466016>)

实现，Task 注解实现方式，比较简单。Quartz 需要手动配置 Jobs。

任务执行，Task 默认单线程串行执行任务，多任务时若某个任务执行时间过长，后续任务会无法及时执行。Quartz 采用多线程，无这个问题。

调度，Task 采用顺序执行，若当前调度占用时间过长，下一个调度无法及时执行；

Quartz 采用异步，下一个调度时间到达时，会另一个线程执行调度，不会发生阻塞问题，但调度过多时可能导致数据处理异常

部署，Quartz 可以采用集群方式，分布式部署到多台机器，分配执行定时任务。

九. 乐优商城

1. 项目介绍

乐优商城是一个基于 微服务 架构的、全品类的 B2C 电商购物网站，整个项目分为后台管理和前台门户两个入口。用户可以在前台门户进行商品的浏览、搜索、购物车、下单、支付和个人管理等操作。在后台管理中管理员可以对商品进行管理、制定促销活动、监控商品销售状态与订单进展情况等。

1.1 什么是微服务

（ Martin Fowler 提出的微服务 ）微服务架构就是将单一程序开发成一个微服务，每个微服务运行在自己的进程中，并使用轻量级的机制通信，通常是 HTTP RESTFUL API。这些服务围绕业务能力来划分，并通过自动化部署机制来独立部署。这些服务可以使用不同的编程语言，不同数据库，以保证最低限度的集中式管理。

总结来说：微服务，又称微服务架构，是一种分布式的系统。就是将一个单体架构的应用按业务划分为一个个独立运行的程序即服务，它们之间通过 HTTP 协议进行通信，可以采用不同的编程语言，使用不同的存储技术，自动化部署减

少人为控制，降低出错概率。服务数量越多，管理起来越复杂，因此采用集中化管理。

1.2 微服务有什么优势

独立开发：所有微服务都可以根据各自的功能轻松开发。

独立部署：基于其服务，可以在任何应用程序中单独部署它们。

故障隔离：即使应用程序的一项服务不起作用，系统仍可继续运行。

混合技术：可以使用不同的语言、技术来构建同一应用程序的不同服务。

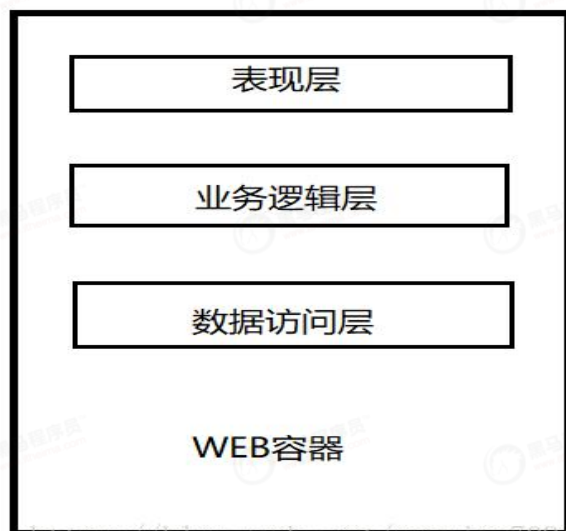
粒度缩放：单个组件可根据需要进行缩放，无需将所有组件缩放在一起。

1.3 SOA 与微服务的关系

SOA 即面向服务的架构，SOA 是根据企业服务总线（ESB）模式来整合集成大量单一庞大的系统。微服务可以说是 SOA 的一种实现，将复杂的业务组件化，但它比 ESB 实现的 SOA 更加的轻便敏捷和简单。

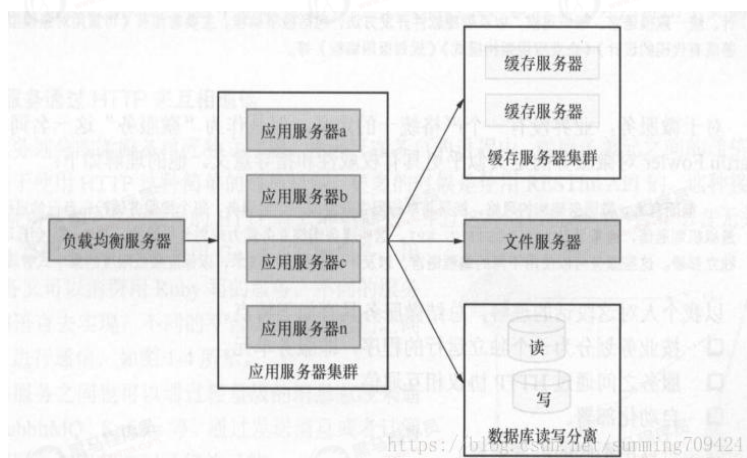
1.4 什么是单体架构

在软件设计的时候经常提到和使用经典的 3 层模型，即表现层，业务逻辑层，数据访问层。虽然在软件设计中划分了 3 层模型，但是对业务场景没有划分，一个典型的单体架构就是将所有的业务场景的表现层，业务逻辑层，数据访问层放在一个工程中最终经过编译，打包，部署在一台服务器上。



1.5 单体架构的不足

在小型应用的初期，访问量小的时候这种架构的性价比还是比较高的，开发速度快，成本低。但是随着业务的发展，逻辑越来越复杂，代码量越来越大，代码的可读性和可维护性越来越低。用户的增加，访问量的越来越多，单体架构的应用并发能力十分有限。可能会有人想到将单体应用进行集群部署，并增加负载均衡服务器，再来个缓存服务器和文件服务器，数据库再搞个读写分离。这种架构如图：



这种架构虽然有一定的并发能力，以及应对一定的复杂业务，但是依然没有改变系统为单体架构的事实。大量的业务必然会有大量的代码，代码的可读性和可维护性依然很差。如果面对海量的用户，它的并发能力依然不够。

1.6 B2C

1.6.1 什么是 B2C

B2C企业对个人,是Business-to-Customer的缩写,中文简称“商对客”。

“商对客”是电子商务的一种模式,也就是通常说的直接面向消费者销售产品和服务的商业零售模式。这种形式的电子商务一般以网络零售业为主,主要是借助于互联网开展在线销售活动。B2C即企业通过互联网为消费者提供一个新型的购物环境——网上商店,消费者通过网络在网上购物、网上支付等消费行为。

案例：唯品会、乐蜂网

1.6.2 其它的电商模式有哪些？

1.B2B—企业对企业

B2B (Business to Business)是指进行电子商务交易的供需双方都是商家 (或企业、公司), 她 (他) 们使用了互联网的技术或各种商务网络平台, 完成商务交易的过程。

案例：阿里巴巴、慧聪网

2.C2C—一个人对个人

C2C 即 Customer (Consumer) to Customer (Consumer), 意思就是消费者个人间的电子商务行为。比如一个消费者有一台电脑,通过网络进行交易,把它出售给另外一个消费者,此种交易类型就称为 C2C 电子商务。

案例：淘宝、易趣、瓜子二手车

3.B2B2C—企业对企业对个人

B2B2C 是一种电子商务类型的网络购物商业模式，B 是 BUSINESS 的简称，C 是 CUSTOMER 的简称，第一个 B 指的是商品或服务的供应商，第二个 B 指的是从事电子商务的企业，C 则表示消费者。

第一个 BUSINESS，并不仅仅局限于品牌供应商、影视制作公司和图书出版商，任何的商品供应商或服务供应商都能可以成为第一个 BUSINESS；第二 B 是 B2B2C 模式的电子商务企业，通过统一的经营管理对商品和服务、消费者终端同时进行整合，是广大供应商和消费者之间的桥梁，为供应商和消费者提供优质的服务，是互联网电子商务服务供应商。C 表示消费者，在第二个 B 构建的统一电子商务平台购物的消费者；

B2B2C 的来源于目前的 B2B、B2C 模式的演变和完善，把 B2C 和 C2C 完美地结合起来，通过 B2B2C 模式的电子商务企业构建自己的物流供应链系统，提供统一的服务。

案例：京东商城、天猫商城

4.O2O—线上对线下

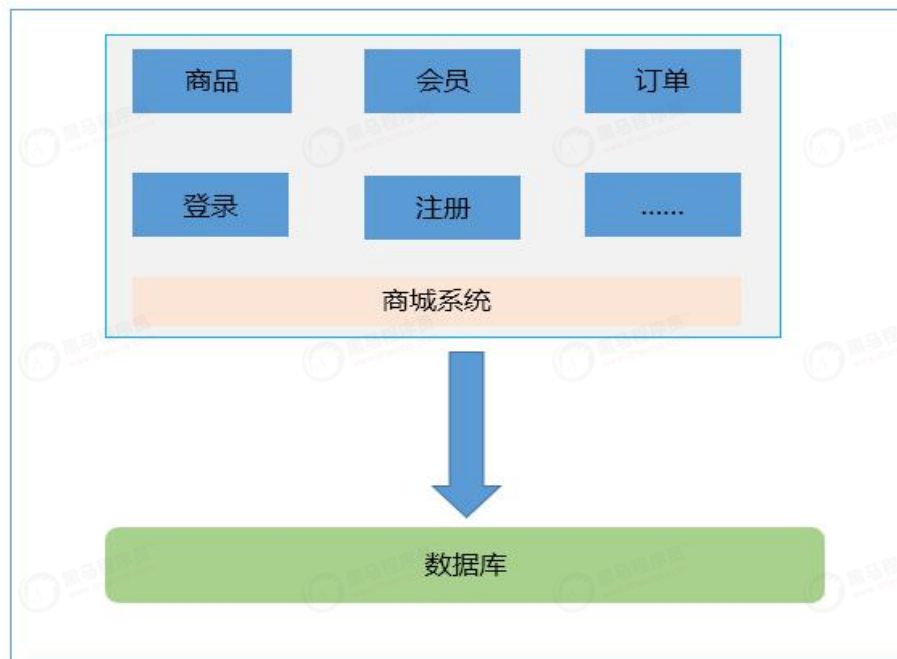
O2O 即 Online To Offline（在线离线/线上到线下），是指将线下的商务机会与互联网结合，让互联网成为线下交易的平台，这个概念最早来源于美国。

O2O 的概念非常广泛，既可涉及到线上，又可涉及到线下，可以通称为 O2O。主流商业管理课程均对 O2O 这种新型的商业模式有所介绍及关注。

案例：美团、饿了么

2. 项目架构

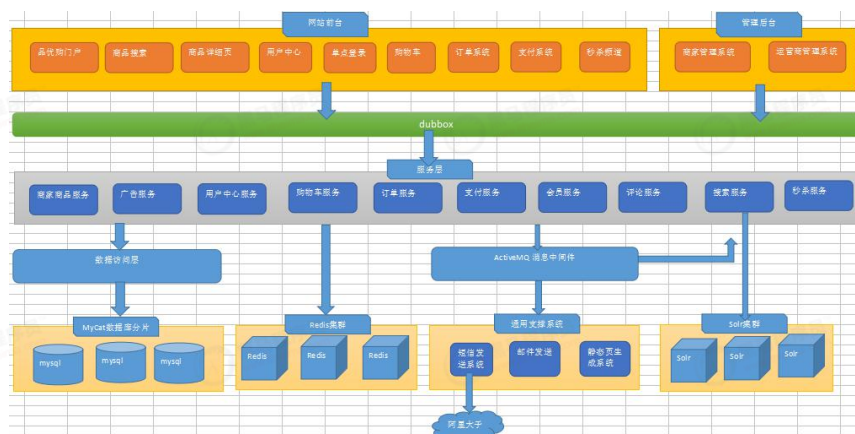
2.1 传统架构



存在的问题

- 1、开发、维护代码比较困难
- 2、系统内部的功能模块之间的耦合度太高了
- 3、无法针对每个模块的特性做优化
- 4、无法做服务器的水平扩展
- 5、无法存储海量数据

2.2 SOA 架构

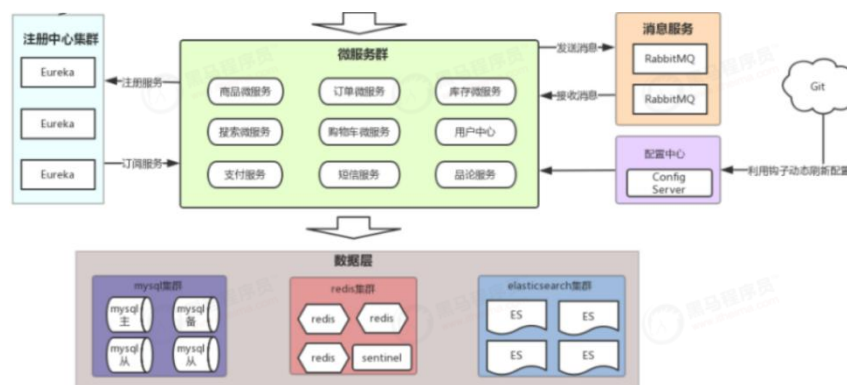


SOA[组装服务/ESB 企业服务总线]（解决了传统架构的所有问题）

业务系统分解为多个组件，让每个组件都独立提供离散，自治，可复用的服务能力。通过服务的组合和编排来实现上层的业务流程。

作用：简化维护，降低整体风险，伸缩灵活。

2.3 微服务架构



微服务，又称微服务架构，是一种分布式的系统（解决了传统架构的所有问题）

3. 项目功能



3.1 人员配置

产品经理：3 人（1 个经理，2 个产品），确定需求以及给出产品原型图。

项目经理：1 人，项目管理。

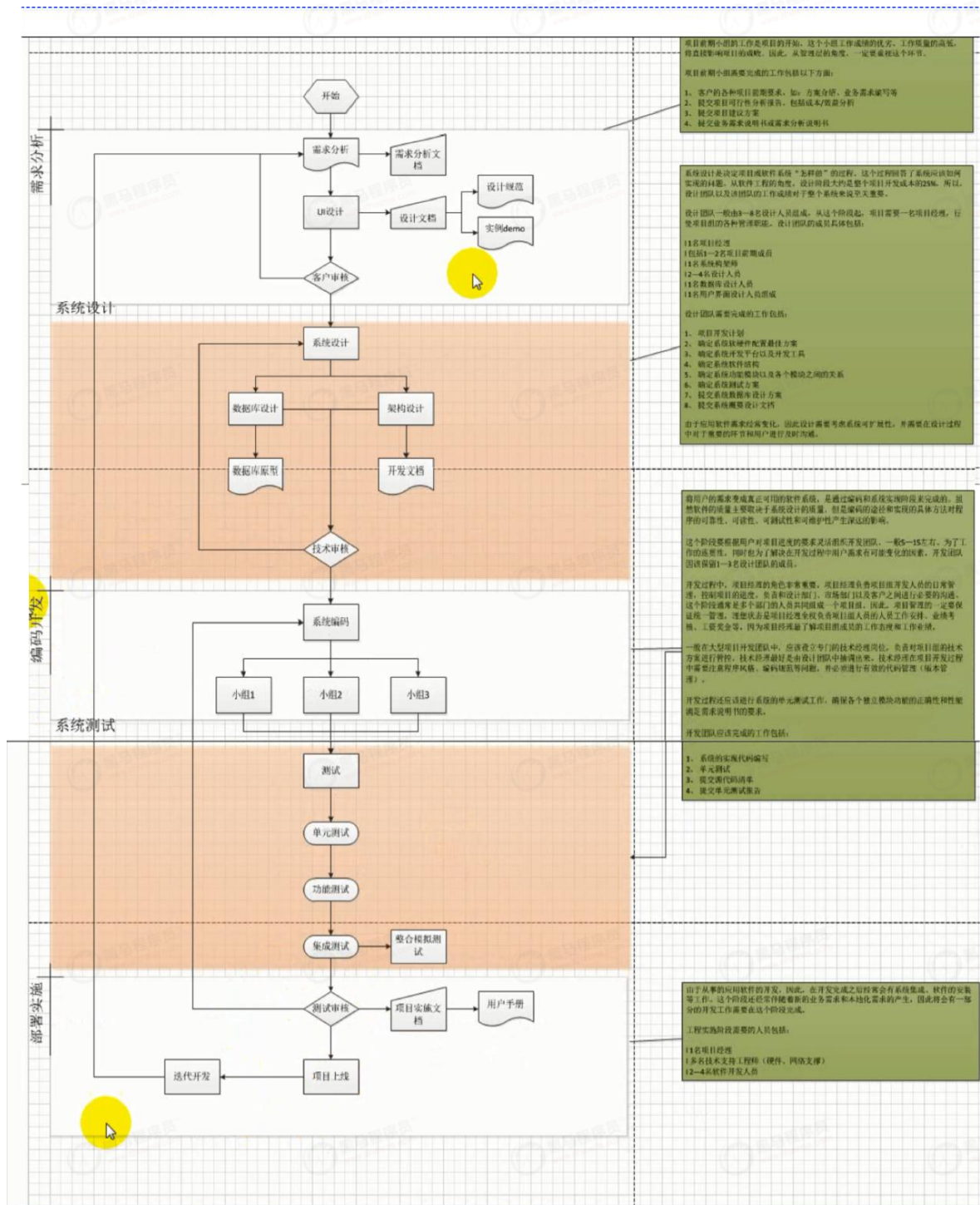
前端团队：2 人，根据产品经理给出的原型图制作静态页面。

后端团队：5 人，根据项目经理分配的任务完成产品功能。

测试团队：2 人，采用白盒测试以及黑盒测试对产品功能性能进行检测。

运维团队：1 人，项目的发布以及维护。

3.2 开发流程



3.3 开发技术

前端

基础的 HTML、CSS、JavaScript (基于 ES6 标准)

JQuery

Vue.js 2.0

基于 Vue 的 UI 框架 : Vuetify

前端构建工具 : WebPack , 项目编译、打包工具

前端安装包工具 : NPM

Vue 脚手架 : Vue-cli

Vue 路由 : vue-router

ajax 框架 : axios

基于 Vue 的富文本框架 : quill-editor

后端

基础的 SpringMVC、Spring 5.0 和 MyBatisPlus

MySQL 5.7

SpringBoot 2.1.3 版本

SpringCloud 版本 Greenwich.Release

Redis-4.0

RabbitMQ-3.7

Elasticsearch-6.2.4

nginx-1.10.2

Thymeleaf 模板语言

JWT

3.4 数据库表结构

表名称	含义
-----	----

tb_application	服务信息表
tb_application_privilege	服务中间表
tb_brand	品牌表
tb_category	商品分类表
tb_category_brand	商品分类和品牌的中间表
tb_order	订单表
tb_order_detail	订单详情表
tb_order_logistics	订单物流表
tb_sku	sku 表
tb_spec_group	规格参数的分组表
tb_spec_param	规格参数组下的参数名表
tb_spu	spu 表
tb_spu_detail	Spu 详情表
tb_user	用户表

4. 模块业务

4.1 业务思路

首先描述自己所做项目有哪些功能模块

举例：最近做了一个电商项目，我负责的模块包括前台系统，搜索系统，还有订单系统。描述其中单个模块有哪些功能

举例：首先前台系统的大部分功能类似于淘宝京东的首页，主要的功能有左侧商品类目的回显，广告位，楼层数据的回显，以及完成这些数据在后台的管理，包

括页面静态化。拆分单独功能进行描述

举例：其中左侧商品类目回显的功能类似于淘宝京东的左侧类目，在首页的左侧有很多商品的一级目录，比如手机，电脑之类的。鼠标放在一级目录上会触发一个事件弹出一个窗口显示二级目录，以此类推总共有三级目录。

描述功能中碰到的问题

举例：在做这个商品类目回显的时候我们发现数据从后台抓取过来，前台页面解析不了。介绍技术，对比技术

举例：对于商品类目数据解析不了的问题是因为跨域，对于跨域我们先是采用了 jsonp 来解决，它的优点是什么？缺点是什么？由于这个技术的缺点我们引入了新的技术，httpclient，这个技术是什么？优点是什么？缺点是什么？介绍完技术，再介绍功能，循环嵌套

5 . 商品管理

5.1 表设计

一件商品的录入大致分为 6 个部分，8 张表：

分类 → tb_category

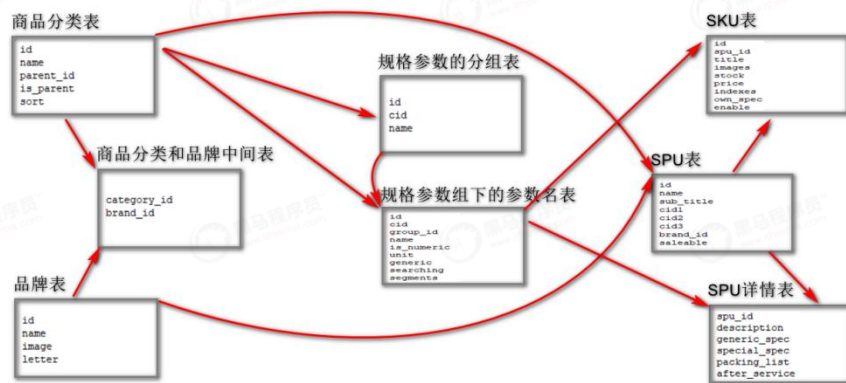
品牌 → tb_brand , tb_category_brand

规格 → tb_spec_group , tb_spec_param

SPU → tb_spu

SKU → tb_sku

商品描述 → tb_spu_detail



5.2 什么是 SPU 和 SKU

SPU : Standard Product Unit (标准产品单位) , 一组具有共同属性的商品集。

SKU : Stock Keeping Unit (库存量单位) , SPU 商品集因具体特性不同而细分的每个商品。

5.3 为什么不设置外键约束？

外键会严重影响数据库读写的效率。

数据删除时会比较麻烦。

在电商行业,性能是非常重要的。我们宁可在代码中通过逻辑来维护表关系,也不设置外键。

5.4 业务

商品管理下有四个小模块,分别是分类列表、品牌列表,规格列表和商品列表。每个模块都分别具有列表、新增、修改和删除等基本功能,其中商品模块不光具有这些功能,还包括商品的上架和下架。

在品牌、商品的新增和修改中值得说的是图片上传。图片上传我们最初采用

的是本地上传：将用户上传的图片保存到 Nginx 的静态资源文件夹下，并生成该图片的访问路径保存到数据库，之后页面通过图片访问路径直接获取 Nginx 下的图片。这种本地图片上传能够实现图片上传功能，但是考虑到单机器存储能力有限，多台机器间无法共享资源，数据没有备份，并发能力差等问题，修改本地图片上传为阿里云存储对象 OSS。阿里的 OSS 是一个提供海量，安全，成本低，高可靠的云存储服务，可以使用 RESTful API 在互联网任何位置存储和访问。用户每次上传图片都会带着后台生产的签名将图片保存到 OSS 中，成功后 OSS 会返回图片的公共访问路径，页面通过此路径访问 OSS 上的图片。

一件商品只有在上架后才能进行销售，在下架后才能对商品的信息进行修改。同时在商品的上架和下架时会通过 RabbitMQ 通知静态页面微服务和搜索微服务进行静态页面的生产和删除，索引库的创建和删除。

5.5 什么是跨域

跨域是指跨域名的访问，以下情况都属于跨域：

域名相同，端口不同：www.jd.com:8080 与 www.jd.com:8081。

域名不同：www.jd.com 与 www.taobao.com。

二级域名不同：item.jd.com 与 miaosha.jd.com。

如果域名和端口都相同，但是请求路径不同，不属于跨域，如：

www.jd.com/item 与 www.jd.com/goods。

5.6 为什么有跨域问题

跨域不一定会有跨域问题。因为跨域问题是浏览器对于 ajax 请求的一种安

全限制：一个页面发起的 ajax 请求，只能是从当前页同域名的路径，这能有效阻止跨站攻击。

因此：跨域问题是针对 ajax 的一种限制。

但是这却给我们的开发带来了不变，而且在实际生成环境中，肯定会有很多台服务器之间交互，地址和端口都可能不同，怎么办？

5.7 解决跨域问题的方案

目前比较常用的跨域解决方案有 3 种：

1. Jsonp：最早的解决方案，使利用 script 标签可以跨域的原理实现。

缺点：需要服务的支持，只能发起 GET 请求。

2. nginx 反向代理：利用 nginx 反向代理把跨域置成不跨域，支持各种请求方式。

缺点：需要在 nginx 进行额外配置，语义不清晰。

3. CORS：规范化的跨域请求解决方案，安全可靠。

优点：在服务端进行控制是否允许跨域，可自定义规则。支持各种请求方式。

缺点：会产生额外的请求。

我们这里会采用 cors 的跨域方案。

5.8 CORS

5.8.1 什么使 CORS

CORS 是一个 W3C 标准，全称是"跨域资源共享" (Cross-origin resource

sharing)。它允许浏览器向跨源服务器，发出 XMLHttpRequest 请求，从而克服了 AJAX 只能同源使用的限制。CORS 需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE 浏览器不能低于 IE10。

浏览器端：

目前，所有浏览器都支持该功能（IE10 以下不行）。整个 CORS 通信过程，都是浏览器自动完成，不需要用户参与。

服务器端：

CORS 通信与 AJAX 没有任何差别，因此你不需要改变以前的业务逻辑。只不过，浏览器会在请求中携带一些头信息，我们需要以此判断是否运行其跨域，然后在响应头中加入一些信息即可。这一般通过过滤器完成即可。

5.8.2 CORS 原理

浏览器会将 ajax 请求分为两类，其处理方案略有差异：简单请求、特殊请求。

简单请求

只要同时满足以下两大条件，就属于简单请求：

1. 请求方法是以下三种方法之一：

HEAD

GET

POST

2. HTTP 的头信息不超出以下几种字段：

Accept

Accept-Language

Content-Language

Last-Event-ID

Content-Type : 只限于三个值

application/x-www-form-urlencoded

multipart/form-data 、 text/plain

当浏览器发现发现的 ajax 请求是简单请求时，会在请求头中携带一个字段：

Origin



```
Request Headers
⚠ Provisional headers are shown
Accept: application/json, text/plain, */*
Origin: http://manage.leyou.com
Referer: http://manage.leyou.com/
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/7.36
```

Origin 中会指出当前请求属于哪个域（协议+域名+端口）。服务会根据这个值决定是否允许其跨域。如果服务器允许跨域，需要在返回的响应头中携带下面信息



```
Access-Control-Allow-Origin: http://manage.leyou.com
Access-Control-Allow-Credentials: true
```

Access-Control-Allow-Origin：可接受的域，是一个具体域名或者*，代表任意。

Access-Control-Allow-Credentials：是否允许携带 cookie，默认情况下，cors 不会携带 cookie，除非这个值是 true。

注意：如果跨域请求要想操作 cookie，需要满足 3 个条件：

服务的响应头中需要携带 Access-Control-Allow-Credentials 并且为 true。

浏览器发起 ajax 需要指定 withCredentials 为 true。

响应头中的 Access-Control-Allow-Origin 一定不能为*，必须是指定的域名。

特殊请求

不符合简单请求的条件，会被浏览器判定为特殊请求，例如请求方式为 PUT。

预检请求

特殊请求会在正式通信之前，增加一次 HTTP 查询请求，称为“预检”请求（preflight）。浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些 HTTP 动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的 XMLHttpRequest 请求，否则就报错。一个“预检”请求的样板：

```
OPTIONS /cors HTTP/1.1
Origin: http://manage.leyou.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-Custom-Header
Host: api.leyou.com
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

与简单请求相比，除了 Origin 以外，多了两个头：

Access-Control-Request-Method：接下来会用到的请求方式，比如 PUT。

Access-Control-Request-Headers：会额外用到的头信息。

预检请求的响应

服务的收到预检请求，如果许可跨域，会发出响应：

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://manage.leyou.com
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: X-Custom-Header
Access-Control-Max-Age: 1728000
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

除了 Access-Control-Allow-Origin 和 Access-Control-Allow-Credentials

以外，这里又额外多出 3 个头：

Access-Control-Allow-Methods：允许访问的方式。

Access-Control-Allow-Headers：允许携带的头。

Access-Control-Max-Age：本次许可的有效时长，单位是秒，过期之前的 ajax 请求就无需再次进行预检了。

如果浏览器得到上述响应，则认定为可以跨域，后续就跟简单请求的处理是一样的了。

5.8.3 实现

虽然原理比较复杂，但是前面说过：

浏览器端都有浏览器自动完成，我们无需操心服务端可以通过拦截器统一实现，不必每次都去进行跨域判定的编写。

在实现代码我们需要做的事情如下：

创建 cors 的配置信息

允许访问的域

是否允许发送 cookie

允许的请求方式

允许的头信息

访问有效期

添加映射路径，我们拦截一切请求。

返回新的 CorsFilter。

事实上，SpringMVC 已经帮我们写好了 CORS 的跨域过滤器：CorsFilter，内部已经实现了刚才所讲的判定逻辑，我们直接用就好了。

5.9 为什么图片地址需要使用另外的 url？

图片不能保存在服务器内部，这样会对服务器产生额外的加载负担。

一般静态资源都应该使用独立域名，这样访问静态资源时不会携带一些不必要的 cookie，减小请求的数据量。

5.10 消息中间件

5.10.1 什么是消息中间件

消息队列已经逐渐成为企业 IT 系统内部通信的核心手段。它具有低耦合、可靠投递、广播、流量控制、最终一致性等一系列功能，成为异步 RPC 的主要手段之一。当今市面上有很多主流的消息中间件，如老牌的 ActiveMQ、RabbitMQ，炙手可热的 Kafka，阿里巴巴自主开发 RocketMQ 等。

5.10.2 消息中间件的组成

Broker

消息服务器，作为 server 提供消息核心服务。

Producer

消息生产者，业务的发起方，负责生产消息传输给 broker。

Consumer

消息消费者，业务的处理方，负责从 broker 获取消息并进行业务逻辑处理。

Topic

主题，发布订阅模式下的消息统一汇集地，不同生产者向 topic 发送消息，由 MQ 服务器分发到不同的订阅者，实现消息的广播。

Queue

队列，PTP 模式下，特定生产者向特定 queue 发送消息，消费者订阅特定的 queue 完成指定消息的接收。

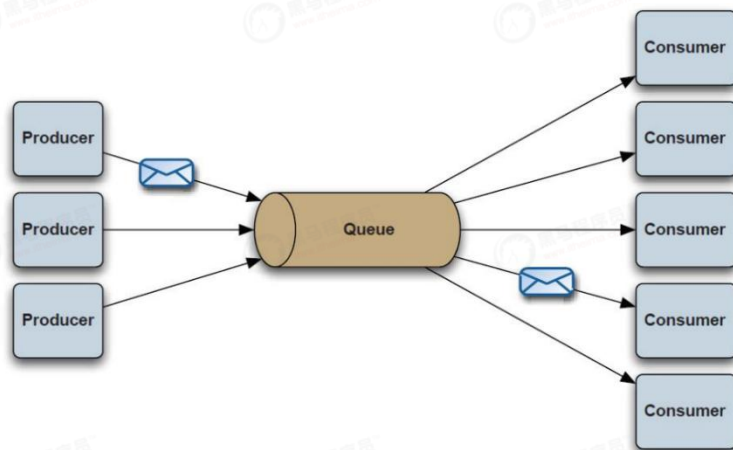
Message

消息体，根据不同通信协议定义的固定格式进行编码的数据包，来封装业务数据，实现消息的传输。

5.10.3 消息中间件模式分类

点对点

PTP 点对点：使用 queue 作为通信载体。



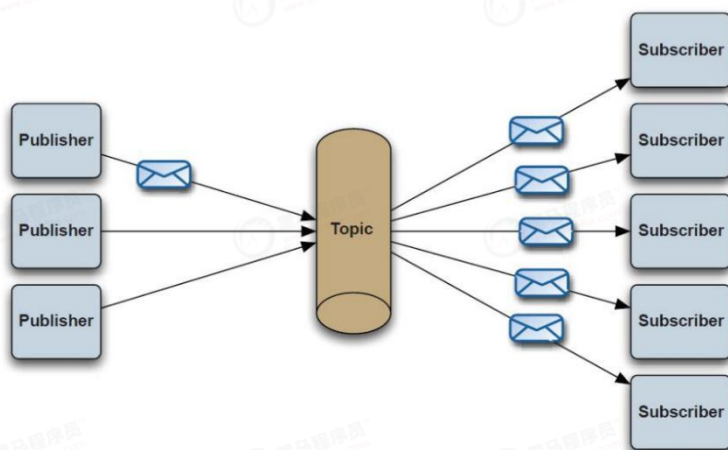
说明：

消息生产者生产消息发送到 queue 中，然后消息消费者从 queue 中取出并且消费消息。

消息被消费以后，queue 中不再存储，所以消息消费者不可能消费到已经被消费的消息。 Queue 支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。

发布/订阅

Pub/Sub 发布订阅（广播）：使用 topic 作为通信载体。



说明：

消息生产者（发布）将消息发布到 topic 中，同时有多个消息消费者（订阅）消费该消息。和点对点方式不同，发布到 topic 的消息会被所有订阅者消费。

queue 实现了负载均衡，将 producer 生产的消息发送到消息队列中，由多个消费者消费。但一个消息只能被一个消费者接受，当没有消费者可用时，这个消息会被保存直到有一个可用的消费者。

topic 实现了发布和订阅，当你发布一个消息，所有订阅这个 topic 的服务都能得到这个消息，所以从 1 到 N 个订阅者都能得到一个消息的拷贝。

5.10.4 消息中间件的优势

系统解耦

交互系统之间没有直接的调用关系，只是通过消息传输，故系统侵入性不强，耦合度低。

提高系统响应时间

例如原来的一套逻辑，完成支付可能涉及先修改订单状态、计算会员积分、通知物流配送几个逻辑才能完成；通过 MQ 架构设计，就可将紧急重要（需要立刻响应）的业务放到该调用方法中，响应要求不高的使用消息队列，放到 MQ 队列中，供消费者处理。

为大数据处理架构提供服务

通过消息作为整合，大数据的背景下，消息队列还与实时处理架构整合，为数据处理提供性能支持。

Java 消息服务——JMS

Java 消息服务（Java Message Service，JMS）应用程序接口是一个 Java 平台中关于面向消息中间件（MOM）的 API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

JMS 中的 P2P 和 Pub/Sub 消息模式：点对点（point to point，queue）与发布订阅（publish/subscribe，topic）最初是由 JMS 定义的。这两种模式主要区别或解决的问题就是发送到队列的消息能否重复消费(多订阅)。

5.10.5 消息中间件常用协议

AMQP 协议

AMQP 即 Advanced Message Queuing Protocol，一个提供统一消息服务的应用层标准高级消息队列协议,是应用层协议的一个开放标准，为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同开发语言等条件的限制。

优点：可靠、通用。

MQTT 协议

MQTT（Message Queuing Telemetry Transport，消息队列遥测传输）是 IBM 开发的一个即时通讯协议，有可能成为物联网的重要组成部分。该协议支持所有平台，几乎可以把所有联网物品和外部连接起来，被用来当做传感器和致动器（比如通过 Twitter 让房屋联网）的通信协议。

优点：格式简洁、占用带宽小、移动端通信、PUSH、嵌入式系统。

STOMP 协议

STOMP（Streaming Text Orientated Message Protocol）是流文本定向消息协议，是一种为 MOM(Message Oriented Middleware，面向消息的中间件)设计的简单文本协议。STOMP 提供一个可互操作的连接格式，允许客户端与任意 STOMP 消息代理（Broker）进行交互。

优点：命令模式（非 topic\queue 模式）。

XMPP 协议

XMPP（可扩展消息处理现场协议，Extensible Messaging and Presence Protocol）是基于可扩展标记语言（XML）的协议，多用于即时消息（IM）以及在线现场探测。适用于服务器之间的准即时操作。核心是基于 XML 流传输，这个协议可能最终允许因特网用户向因特网上的其他任何人发送即时消息，即使其操作系统和浏览器不同。

优点：通用公开、兼容性强、可扩展、安全性高，但 XML 编码格式占用带宽大。

其他基于 TCP/IP 自定义的协议

有些特殊框架（如：Redis、kafka、zeroMq 等）根据自身需要未严格遵循 MQ 规范，而是基于 TCP/IP 自行封装了一套协议，通过网络 socket 接口进行传输，实现了 MQ 的功能。

5.11 RabbitMQ

5.11.1 什么是 RabbitMQ？

RabbitMQ 是一款开源的，Erlang 编写的，基于 AMQP 协议的，消息中间件。

5.11.2 为什么要使用 RabbitMQ？Rabbit 有什么优点？

解耦、异步、削峰。

5.11.3 RabbitMQ 有什么缺点？

降低了系统的稳定性：本来系统运行好好的，现在你非要加入个消息队列进去，那消息队列挂了，你的系统不是呵呵了。因此，系统可用性会降低。

增加了系统的复杂性：加入了消息队列，要多考虑很多方面的问题，比如：一致性问题、如何保证消息不被重复消费、如何保证消息可靠性传输等。因此，需要考虑的东西更多，复杂性增大。

5.11.4 RabbitMQ 的工作模式？

简单模式

一个生产者，一个消费者。

work 模式

一个生产者，多个消费者，每个消费者获取到的消息唯一。

订阅模式

一个生产者发送的消息会被多个消费者获取。

路由模式

发送消息到交换机并且要指定路由 key，消费者将队列绑定到交换机时需要指定路由 key。

topic 模式

将路由键和某模式进行匹配，此时队列需要绑定在一个模式上，“#”匹配一个词或多个词，“*”只匹配一个词。

5.11.5 如何保证 RabbitMQ 高可用？

搭建 RabbitMQ 集群。

5.11.6 如何保证 RabbitMQ 消息不被重复消费？

先说为什么会重复消费：正常情况下，消费者在消费消息的时候，消费完毕后，会发送一个确认消息给消息队列，消息队列就知道该消息被消费了，就会将该消息从消息队列中删除；

但是因为网络传输等等故障，确认信息没有传送到消息队列，导致消息队列不知道自己已经消费过该消息了，再次将消息分发给其他的消费者。

针对以上问题，一个解决思路是：保证消息的唯一性，就算是多次传输，不要让消息的多次消费带来影响，保证消息幂等性。

比如：在写入消息队列的数据做唯一标示，消费消息时，根据唯一标识判断是否消费过。

5.11.7 如何保证 RabbitMQ 消息可靠传输？

消息不可靠的情况可能是消息丢失，劫持等原因。丢失又分为：生产者丢失消息、消息列表丢失消息、消费者丢失消息。

生产者丢失消息

从生产者弄丢数据这个角度来看，RabbitMQ 提供 transaction 和 confirm 模式来确保生产者不丢消息。

transaction 模式：发送消息前，开启事务 (`channel.txSelect()`) ,然后发送

消息，如果发送过程中出现什么异常，事务就会回滚（`channel.txRollback()`），如果发送成功则提交事务（`channel.txCommit()`）。然而，这种方式有个缺点，吞吐量下降。

confirm 模式：一旦 channel 进入 confirm 模式，所有在该信道上发布的消息都将会被指派一个唯一的 ID（从 1 开始），一旦消息被投递到所有匹配的队列之后。RabbitMQ 就会发送一个 ACK 给生产者（包含消息的唯一 ID），这就使得生产者知道消息已经正确到达目的队列了。如果 rabbitMQ 没能处理该消息，则会发送一个 Nack 消息给你，你可以进行重试操作。

消息队列丢数据

消息持久化。处理消息队列丢数据的情况，一般是开启持久化磁盘的配置。这个持久化配置可以和 confirm 机制配合使用，你可以在消息持久化磁盘后，再给生产者发送一个 Ack 信号。这样，如果消息持久化磁盘之前，rabbitMQ 阵亡了，那么生产者收不到 Ack 信号，生产者会自动重发。那么如何持久化呢？这里顺便说一下，其实也很容易，就下面两步：

1. 将 queue 的持久化标识 `durable` 设置为 `true`，则代表是一个持久的队列。
2. 发送消息的时候将 `deliveryMode=2`。

这样设置以后，即使 rabbitMQ 挂了，重启后也能恢复数据。

消费者丢失消息

消费者丢数据一般是因为采用了自动确认消息模式，改为手动确认消息即可！消费者在收到消息之后，处理消息之前，会自动回复 RabbitMQ 已收到消息。如果这时处理消息失败，就会丢失该消息。

解决方案：处理消息成功后，手动回复确认消息。

5.11.8 如何保证 RabbitMQ 消息的顺序性？

单线程消费保证消息的顺序性。对消息进行编号，消费者处理消息是根据编号处理消息。

5.11.9 如何确保消息正确地发送至 RabbitMQ？如何确保消息接收方消费了消息？

发送方确认模式

将信道设置成 confirm 模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的 ID。一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一 ID）。如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack（not acknowledged，未确认）消息。发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。

接收方确认机制

消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。这里并没有用到超时机制，RabbitMQ 仅通过 Consumer 的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ 给了 Consumer 足够长的时间来处理消息。保证数据的最终一致性。

下面罗列几种特殊情况

如果消费者接收到消息，在确认之前断开了连接或取消订阅，RabbitMQ 会认为消息没有被分发，然后重新分发给下一个订阅的消费者。（可能存在消息重

复消费的隐患，需要去重）。

如果消费者接收到消息却没有确认消息，连接也未断开，则 RabbitMQ 认为该消费者繁忙，将不会给该消费者分发更多的消息。

5.11.10 RabbitMQ 消息堆积处理

1. 增加消费者的处理能力(例如优化代码)，或减少发布频率。
2. 考虑使用队列最大长度限制。
3. 给消息设置年龄，超时就丢弃。
4. 默认情况下，rabbitmq 消费者为单线程串行消费，设置并发消费两个关键属性 `concurrentConsumers` 和 `prefetchCount` 设置的是对每个 listener 在初始化的时候设置的并发消费者的个数，`prefetchCount` 是每次一次性从 broker 里面取的待消费的消息的个数。
5. 建立新的 queue，消费者同时订阅新旧 queue，采用订阅模式。
6. 生产者端缓存数据，在 mq 被消费完后再发送到 mq，打破发送循环条件，设置合适的 qos 值，当 qos 值被用光，而新的 ack 没有被 mq 接收时，就可以跳出发送循环，去接收新的消息；消费者主动 block 接收进程，消费者感受到接收消息过快时主动 block，利用 `block` 和 `unblock` 方法调节接收速率，当接收线程被 block 时，跳出发送循环。

5.11.11 RabbitMQ 消息丢失解决方案

消息持久化

Exchange 设置持久化：`durable:true`。

ueue 设置持久化；Message 持久化发送。

ACK 确认机制

消息发送确认。

息接收手动确认 ACK。

5.11.12 RabbitMQ 宕机了怎么处理

RabbitMQ 提供了持久化的机制，将内存中的消息持久化到硬盘上，即使重启 RabbitMQ，消息也不会丢失。

持久化队列和非持久化队列的区别是，持久化队列会被保存在磁盘中，固定并持久的存储，当 Rabbit 服务重启后，该队列会保持原来的状态在 RabbitMQ 中被管理，而非持久化队列不会被保存在磁盘中，Rabbit 服务重启后队列就会消失。

非持久化比持久化的优势就是，由于非持久化不需要保存在磁盘中，所以使用速度就比持久化队列快。即是非持久化的性能要高于持久化。

而持久化的优点就是会一直存在，不会随服务的重启或服务器的宕机而消失。

5.11.13 RabbitMQ 的集群

Rabbitmq 有 3 种模式，但集群模式有 2 中，详情如下：

单一模式

即单机情况不做集群，就单独运行一个 rabbitmq 而已。

普通模式

默认模式 ,以两个节点(rabbit01、 rabbit02)为例来进行说明。对于 Queue 来说 , 消息实体只存在于其中一个节点 rabbit01 (或者 rabbit02) , rabbit01 和 rabbit02 两个节点仅有相同的元数据 , 即队列的结构。当消息进入 rabbit01 节点的 Queue 后 , consumer 从节点消费时 , RabbitMQ 会临时在 rabbit01、 rabbit02 间进行消息传输 , 把 A 中的消息实体取出并经过 B 发送给 consumer。所以 consumer 应尽量连接每一个节点 , 从中取消息。即对于同一个逻辑队列 , 要在多个节点建立物理 Queue。否则无论 consumer 连 rabbit01 或 rabbit02 , 出口总在 rabbit01 , 会产生瓶颈。当 rabbit01 节点故障后 , rabbit02 节点无法取到 rabbit01 节点中还未消费的消息实体。如果做了消息持久化 , 那么得等 rabbit01 节点恢复 , 然后才可被消费 ; 如果没有持久化的话 , 就会产生消息丢失的现象。

镜像模式

把需要的队列做成镜像队列 , 存在与多个节点属于 RabbitMQ 的 HA 案。该模式解决了普通模式中的问题 , 其实质和普通模式不同之处在于 , 消息实体会主动在镜像节点间同步 , 而不是在客户端取数据时临时拉取。该模式带来的副作用也很明显 , 除了降低系统性能外 , 如果镜像队列数量过多 , 加之大量的消息进入 , 集群内部的网络带宽将会被这种同步通讯大大消耗掉。所以在对可靠性要求较高的场合中适用。

5.11.14 如何解决分布式事务问题 ? (热点)

1.2PC : 两阶段提交

执行阶段 :

当创建订单时，向所有微服务发送消息,可以开始执行了,执行成功后，不直接提交,而是返回一个消息告诉我,执行成功还是执行失败.

第二阶段：

如果所有人都执行成功,再发一个消息,你们可以提交了

如果第一阶段有人执行失败,你就告诉所有人都回滚

缺点：当锁定的表很多时,性能差,

3.TCC：补偿性事务(一般采用) try-confirm-concel

每个服务执行完后都提交,集中返回给自己,如果都执行成功了那就不管

如果提交失败,就采用失败服务的补偿方法去补偿,但若补偿方法也失败,那你还需要进行重试写重试方法或者人工介入.

优缺：解决了性能问题,但是业务复杂,写一个事务还需要写补偿方法

异步确保：利用 mq 实现分布式事务

5.10.6 常见的消息中间

RocketMQ

阿里系下开源的一款分布式、队列模型的消息中间件，原名 Metaq，3.0 版本名称改为 RocketMQ，是阿里参照 kafka 设计思想使用 java 实现的一套 mq。同时将阿里系内部多款 mq 产品（Notify、metaq）进行整合，只维护核心功能，去除了所有其他运行时依赖，保证核心功能最简化，在此基础上配合阿里上述其他开源产品实现不同场景下 mq 的架构，目前主要多用于订单交易系统。

RabbitMQ

使用 Erlang 编写的一个开源的消息队列，本身支持很多的协议：AMQP，

XMPP, SMTP, STOMP, 也正是如此, 使它变的非常重量级, 更适合于企业级的开发。同时实现了 Broker 架构, 核心思想是生产者不会将消息直接发送给队列, 消息在发送给客户端时先在中心队列排队。对路由(Routing), 负载均衡(Load balance)、数据持久化都有很好的支持。多用于进行企业级的 ESB 整合。

ActiveMQ

Apache 下的一个子项目。使用 Java 完全支持 JMS1.1 和 J2EE 1.4 规范的 JMS Provider 实现, 少量代码就可以高效地实现高级应用场景。可插拔的传输协议支持, 比如: in-VM, TCP, SSL, NIO, UDP, multicast, JGroups and JXTA transports。RabbitMQ、ZeroMQ、ActiveMQ 均支持常用的多种语言客户端 C++、Java、.Net、Python、Php、Ruby 等。

Redis

使用 C 语言开发的一个 Key-Value 的 NoSQL 数据库, 开发维护很活跃, 虽然它是一个 Key-Value 数据库存储系统, 但它本身支持 MQ 功能, 所以完全可以当做一个轻量级的队列服务来使用。对于 RabbitMQ 和 Redis 的入队和出队操作, 各执行 100 万次, 每 10 万次记录一次执行时间。测试数据分为 128Bytes、512Bytes、1K 和 10K 四个不同大小的数据。实验表明: 入队时, 当数据比较小时 Redis 的性能要高于 RabbitMQ, 而如果数据大小超过了 10K, Redis 则慢得无法忍受; 出队时, 无论数据大小, Redis 都表现出非常好的性能, 而 RabbitMQ 的出队性能则远低于 Redis。

Kafka

Apache 下的一个子项目, 使用 scala 实现的一个高性能分布式 Publish/Subscribe 消息队列系统。

5.12 图片上传

本地存储-业务思路

用户在页面选择图片点击上传，到图片微服务，图片微服务解析请求，复制图片并保存到 Nginx 静态资源下（html）。同时生成此图片的访问路径（url）保存到数据库。页面通过 url 访问 Nginx 下的图片。

阿里云对象存储 OSS-业务思路

用户在页面选择图片点击上传，首先访问后台，获取访问 OSS 的签名，请求带着 OSS 的签名将图片上传到 OSS。OSS 成功保存图片后会返回该图片的公共访问路径（url），将此路径保存到数据。页面通过 url 访问 OSS 上的图片

什么是分布式文件系统

分布式文件系统（Distributed File System）是指文件系统管理的物理存储资源不一定直接连接在本地节点上，而是通过计算机网络与节点相连。

通俗来讲

传统文件系统管理的文件就存储在本机。

分布式文件系统管理的文件存储在很多机器，这些机器通过网络连接，要被统一管理。无论是上传或者访问文件，都需要通过管理中心来访问。

常见的分布式文件系统有谷歌的 GFS、HDFS（Hadoop）、TFS（淘宝）、FastDFS（淘宝）等。

不过，企业自己搭建分布式文件系统成本较高，对于一些中小型企业而言，使用云上的文件存储，是性价比更高的选择。

OSS 简介

阿里云对象存储服务（Object Storage Service，简称 OSS），是阿里云提

供的海量、安全、低成本、高可靠的云存储服务。其数据设计持久性不低于 99.999999999% , 服务设计可用性不低于 99.99%。具有与平台无关的 RESTful API 接口 , 您可以在任何应用、任何时间、任何地点存储和访问任意类型的数据。

您可以使用阿里云提供的 API、SDK 接口或者 OSS 迁移工具轻松地将海量数据移入或移出阿里云 OSS。数据存储到阿里云 OSS 以后 , 您可以选择标准类型 (Standard) 的阿里云 OSS 服务作为移动应用、大型网站、图片分享或热点音视频的主要存储方式 , 也可以选择成本更低、存储期限更长的低频访问类型 (Infrequent Access) 和归档类型 (Archive) 的阿里云 OSS 服务作为不经常访问数据的备份和归档。

基本概念

存储类型 (Storage Class)

OSS 提供标准、低频访问、归档三种存储类型 , 全面覆盖从热到冷的各种数据存储场景。其中标准存储类型提供高可靠、高可用、高性能的对象存储服务 , 能够支持频繁的数据访问 ; 低频访问存储类型适合长期保存不经常访问的数据 (平均每月访问频率 1 到 2 次) , 存储单价低于标准类型 ; 归档存储类型适合需要长期保存 (建议半年以上) 的归档数据 , 在三种存储类型中单价最低。详情请参见存储类型介绍。

存储空间 (Bucket)

存储空间是您用于存储对象 (Object) 的容器 , 所有的对象都必须隶属于某个存储空间。您可以设置和修改存储空间属性用来控制地域、访问权限、生命周期等 , 这些属性设置直接作用于该存储空间内所有对象 , 因此您可以通过灵活创建不同的存储空间来完成不同的管理功能。

对象/文件 (Object)

对象是 OSS 存储数据的基本单元，也被称为 OSS 的文件。对象由元信息 (Object Meta)，用户数据 (Data) 和文件名 (Key) 组成。对象由存储空间内部唯一的 Key 来标识。对象元信息是一组键值对，表示了对象的一些属性，比如最后修改时间、大小等信息

地域 (Region)

地域表示 OSS 的数据中心所在物理位置。您可以根据费用、请求来源等综合选择数据存储的地域。详情请参见 OSS 已开通的 Region。

访问域名 (Endpoint)

Endpoint 表示 OSS 对外服务的访问域名。OSS 以 HTTP RESTful API 的形式对外提供服务，当访问不同地域的时候，需要不同的域名。通过内网和外网访问同一个地域所需要的域名也是不同的。具体的内容请参见各个 Region 对应的 Endpoint。

访问密钥 (AccessKey)

AccessKey，简称 AK，指的是访问身份验证中用到的 AccessKeyId 和 AccessKeySecret。OSS 通过使用 AccessKeyId 和 AccessKeySecret 对称加密的方法来验证某个请求的发送者身份。AccessKeyId 用于标识用户，AccessKeySecret 是用户用于加密签名字符串和 OSS 用来验证签名字符串的密钥，其中 AccessKeySecret 必须保密。

5.13 FastDFS

FastDFS 是一个开源的轻量级分布式文件系统，它可以对文件进行管理，功

能包括：文件存储、文件同步、文件访问（文件上传、文件下载）等，解决了大容量存储和负载均衡的问题。特别适合以文件为载体的在线服务，如相册网站、视频网站等等。

组成

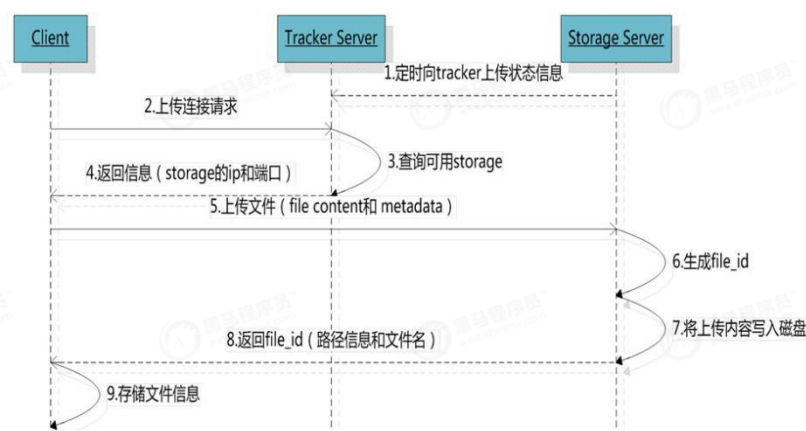
1. Storage server(存储服务器)

Storage server一般都是以组(group)为组织单位，一个组中有多个 Storage server，数据互为备份(意味着每个 Storageserver 的内容是一致的，他们之间没有主从之分)，组的存储空间以组内最小的 Storage server 为准，所以为了避免浪费存储空间最好的话每个 Storage server 的配置最好相同。

2. Tracker server(调度服务器、追踪服务器)

Tracker server 主要负责管理所有的 Storage server 和 group，每个 storage 在启动后会连接 Tracker，告知自己所属的 group 等信息，并保持周期性的心跳，tracker 根据 storage 的心跳信息，建立 group==>[storage server list]的映射表。

流程



1、选择 tracker server

当集群中不止一个 tracker server 时,由于 tracker 之间是完全对等的关系,客户端在 upload 文件时可以任意选择一个 tracker。

2、选择存储的 group

当 tracker 接收到 upload file 的请求时,会为该文件分配一个可以存储该文件的 group,支持如下选择 group 的规则:

1. Round robin, 所有的 group 间轮询。
2. Specified group, 指定某一个确定的 group。
3. Load balance, 剩余存储空间多多 group 优先。
4. 选择 storage server。

3、选择 storage server

当选定 group 后,tracker 会在 group 内选择一个 storage server 给客户端,支持如下选择 storage 的规则:

1. Round robin, 在 group 内的所有 storage 间轮询。
2. First server ordered by ip, 按 ip 排序。
3. First server ordered by priority, 按优先级排序 (优先级在 storage 上配置)。

4、选择 storage path

当分配好 storage server 后,客户端将向 storage 发送写文件请求,storage 将会为文件分配一个数据存储目录,支持如下规则:

1. Round robin, 多个存储目录间轮询。
2. 剩余存储空间最多的优先。

5、生成 Fileid

选定存储目录之后，storage 会为文件生一个 Fileid，由 storage server ip、文件创建时间、文件大小、文件 crc32 和一个随机数拼接而成，然后将这个二进制串进行 base64 编码，转换为可打印的字符串。

6、选择两级目录

当选定存储目录之后，storage 会为文件分配一个 fileid，每个存储目录下有两级 256*256 的子目录，storage 会按文件 fileid 进行两次 hash（猜测），路由到其中一个子目录，然后将文件以 fileid 为文件名存储到该子目录下。

7、生成文件名

当文件存储到某个子目录后，即认为该文件存储成功，接下来会为该文件生成一个文件名，文件名由 group、存储目录、两级子目录、fileid、文件后缀名（由客户端指定，主要用于区分文件类型）拼接而成。

FastDFS 如何现在组内的多个 storage server 的数据同步？

当客户端完成文件写至 group 内一个 storage server 之后即认为文件上传成功，storage server 上传完文件之后，会由后台线程将文件同步至同 group 内其他的 storage server。后台线程同步参考的依据是每个 storageserver 在写完文件后，同时会写一份 binlog，binlog 中只包含文件名等元信息，storage 会记录向 group 内其他 storage 同步的进度，以便重启后能接上次的进度继续同步；进度以时间戳的方式进行记录，所以最好能保证集群内所有 server 的时钟保持同步。

5.14 Nginx

什么

Nginx 是一个高性能的 HTTP 和反向代理服务器，及电子邮件代理服务器，同时也是
一个非常高效的反向代理、负载平衡。

作用

反向代理，将多台服务器代理成一台服务器。

负载均衡，将多个请求均匀的分配到多台服务器上，减轻每台服务器的压力，提高服务的吞吐量。

动静分离，nginx 可以用作静态文件的缓存服务器，提高访问速度。

优势

可以高并发连接（5 万并发，实际也能支持 2~4 万并发）。

内存消耗少。

成本低廉。

配置文件非常简单。

支持 Rewrite 重写。

内置的健康检查功能。

节省带宽。

稳定性高。

支持热部署。

什么是反向代理

反向代理是指以代理服务器来接受 internet 上的连接请求，然后将请求，发给内部网络上的服务器，并将从服务器上得到的结果返回给 internet 上请求连

接的客户端，此时代理服务器对外就表现为一个反向代理服务器。

反向代理总结就一句话：代理端代理的是服务端。

什么是正向代理

一个位于客户端和原始服务器之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)，然后代理向原始服务器转交请求并将获得的内容返回给客户端。客户端才能使用正向代理。

正向代理总结就一句话：代理端代理的是客户端。

负载均衡

负载均衡即是代理服务器将接收的请求均衡的分发到各服务器中，负载均衡主要解决网络拥塞问题，提高服务器响应速度，服务就近提供，达到更好的访问质量，减少后台服务器大并发压力。

Nginx 是如何处理一个请求的

首先，nginx 在启动时，会解析配置文件，得到需要监听的端口与 ip 地址，然后在 nginx 的 master 进程里面先初始化好这个监控的 socket 再进行 listen，然后再 fork 出多个子进程出来，子进程会竞争 accept 新的连接。此时，客户端就可以向 nginx 发起连接了。当客户端与 nginx 进行三次握手，与 nginx 建立好一个连接后，此时，某一个子进程会 accept 成功，然后创建 nginx 对连接的封装，即 ngx_connection_t 结构体，接着，根据事件调用相应的事件处理模块，如 http 模块与客户端进行数据的交换。最后，nginx 或客户端来主动关掉连接，到此，一个连接就寿终正寝了。

为什么 Nginx 性能这么高

得益于它的事件处理机制：异步非阻塞事件处理机制：运用了 epoll 模型，

提供了一个队列，排队解决。

对数据库只是采用了读写分离,并没有完全解决数据库的压力,那么有什么办法解决?

如果数据库压力确实很大的情况下可以考虑数据库分片，就是将数据库中表拆分到不同的数据库中保存。可以使用 mycat 中间件。

商品存入数据库怎么保证数据库数据安全?

对用户安全管理 用户操作数据库时，必须通过数据库访问的身份认证。删除数据库中的默认用户，使用自定义的用户及高强度密码。

定义视图

为不同的用户定义不同的视图，可以限制用户的访问范围。通过视图机制把需要保密的数据对无权存取这些数据的用户隐藏起来，可以对数据库提供一定程度的安全保护。实际应用中常将视图机制与授权机制结合起来使用，首先用视图机制屏蔽一部分保密数据，然后在视图上进一步进行授权。

数据加密 数据加密是保护数据在存储和传递过程中不被窃取或修改的有效手段。

数据库定期备份

审计追踪机制

审计追踪机制是指系统设置相应的日志记录，特别是对数据更新、删除、修改的记录，以便日后查证。日志记录的内容可以包括操作人员的名称、使用的密码、用户的 IP 地址、登录时间、操作内容等。若发现系统的数据遭到破坏，可以根据日志记录追究责任，或者从日志记录中判断密码是否被盗，以便修改密码，重新分配权限，确保系统的安全。

6. 商品搜索

6.1 简介

搜索模块在我们项目中也是比较重要的一个模块，因为商品的搜索是每个用户在购物消费产生之前不可避免的行为，而且每个用户的搜索方式、需求也不一样，比如有的用户在情人节想买个礼物给女朋友，但是他不知道买什么？这个时候他就会按照“情人节礼物 女”这样的条件来进行搜索，系统会自动匹配符合关键字条件的商品展示给用户；再比如有的用户想给长辈买一个手机，但是他不知道具体哪些品牌会有老人机，所以当他输入“老人机”这个关键字时，系统会自动匹配展示所有符合条件的商品，所以说了这么多，想要完成商品的搜索都离不开非常关键的点，关键字。

中文跟英文不一样，他的组合方式有很多种，比如中国人，它可以拆分成“中、国、中国、中国人、国人、人中”，如何合理的分词对我们的商品搜索至关重要。另外搜索的请求如果都走数据库的话，很可能会将整个应用的性能给拖垮了，所以我们单独搭建了一个搜索服务器来试下搜索服务，那至于采用什么搜索引擎，在起初技术选型的时候我们考虑的是 solr 和 elasticSearch，考虑到后期网站并发量会很大，elasticSearch 的性能比 solr 要高，所以我们采用了 elasticSearch 来作为我们的搜索引擎，另外还同时配置了 IK 分词器来对商品的关键字进行分词。

6.2 功能特点

1. 关键字搜索

将页面提交的关键字提交给后台，再使用 spring data es 提供的模板类 esTempalte 完成搜索。其中对商品的搜索是基于构建的 all 字段完成的匹配查询。all 字段中包含了商品的名称+品牌名称+三级分类。

2. 过滤条件查询

关键字生成搜索结果后还提供了过滤条件的添加，过滤条件的生成是根据搜索结果动态展示的，因为不同的商品因分类的不同对应的参数也不同，过滤条件是在关键字搜索的结果集中，通过聚合查询对结果集进行过滤。过滤条件包含分类，品牌，以及商品的规格参数。其中规格名称和规格值都是动态变化的。规格参数的展示是在用户点击某一个分类条件时，查询该分类对应用于搜索的规格名称以及规格值。在关键字搜索的基础上再点击过滤条件进行进一步的筛选商品数据。

6.3 ElasticSearch

6.3.1 概念

ElasticSearch 是一个分布式的 RESTful 风格的搜索和数据分析引擎。它是基于 Lucene 的，提供了具有 HTTP Web 界面和无架构 JSON 文档的分布式，多租户能力的全文搜索引擎。Elasticsearch 是用 Java 开发的，根据 Apache 许可条款作为开源发布。

6.3.2 作用

1. 分布式的搜索引擎和数据分析引擎。
- 2.全文检索，结构化检索，数据分析。

3.对海量数据进行近实时的处理。

6.3.3 优势

- 1.自动维护数据的分布到多个节点的索引的建立,还有搜索请求分布到多个节点的执行。
- 2.自动维护数据的冗余副本,保证了一旦机器宕机,不会丢失数据。
- 3.装了更多高级的功能,例如聚合分析的功能,基于地理位置的搜索。

6.3.4 Elasticsearch 中的倒排索引是什么?

倒排索引是搜索引擎的核心。搜索引擎的主要目标就是在查找发生搜索条件的文档时提供快速搜索。倒排索引也常被称为反向索引、置入档案或反向档案,是一种索引方法,被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构。通过倒排索引,可以根据单词快速获取包含这个单词的文档列表。

6.3.5 Elasticsearch 中的集群、节点、索引、文档、类型是什么?

群集 是一个或多个节点(服务器)的集合,它们共同保存您的整个数,并提供跨所有节点的联合索引和搜索功能。群集由唯一名称标识,默认情况下为“elasticsearch”。此名称很重要,因为如果节点设置为按名称加入群集,则该节点只能是群集的一部分。

节点 是属于集群一部分的单个服务器。它存储数据并参与群集索引和搜索功能。

索引 就像关系数据库中的“数据库”。它有一个定义多种类型的映射。索引是逻辑名称空间，映射到一个或多个主分片，并且可以有零个或多个副本分片。

MySQL => 数据库，ElasticSearch => 索引。

文档 类似于关系型数据库中的一行。不同之处在于索引中的每个文档可以具有不同的结构(字段)，但是对于通用字段应该具有相同的数据类型。 MySQL => Databases => Tables => Columns / Rows ElasticSearch => Indices => Types => 具有属性的文档。

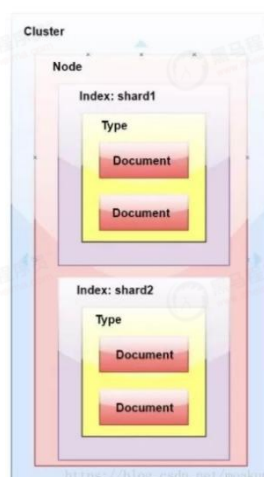
类型 是索引的逻辑类别/分区，其语义完全取决于用户。

6.3.6 ElasticSearch 中的分片是什么？

在大多数环境中，每个节点都在单独的盒子或虚拟机上运行。

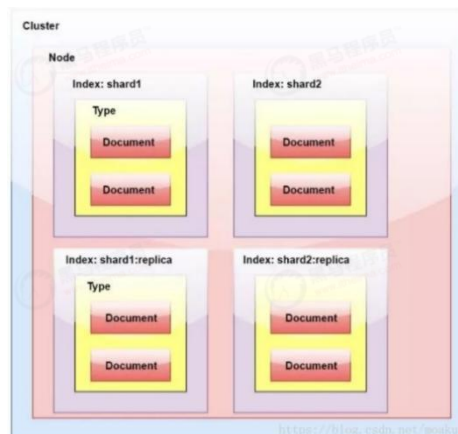
索引 - 在 Elasticsearch 中，索引是文档的集合。

分片 - 因为 Elasticsearch 是一个分布式搜索引擎，所以索引通常被分割成分布在多个节点上的被称为分片的元素。



6.3.7 Elasticsearch 中的副本是什么？

一个索引被分解成碎片以便于分发和扩展。副本是分片的副本。一个节点是一个属于一个集群的 Elasticsearch 的运行实例。一个集群由一个或多个共享相同集群名称的节点组成。



6.3.8.ElasticSearch 查询方式有哪几种？

1. 查询所有(match_all)

2. 匹配查询(match)

会把查询条件进行分词，然后进行查询，多个词条之间默认关系是 OR。

3. 词条查询(term)

用于精确值 匹配，这些精确值可能是数字、时间、布尔或者那些未分词的字符串。

4. 布尔组合(bool)

各种其它查询通过 must（与）、must_not（非）、should（或）的方式进行组合。

5. 范围查询(range)

找出那些落在指定区间内的数字或者时间。

6. 模糊查询(fuzzy)

fuzzy 查询是 term` 查询的模糊等价。它允许用户搜索词条与实际词条的拼写出现偏差，但是偏差的编辑距离不得超过 2。

6.4 Solr

6.4.1 概念

Solr 是基于 Lucene 实现的搜索引擎，扩展性良好，并且提供了完整的集群方案和索引库优化方案。

6.4.2 优势

1. solr 是将整个索引操作功能封装好了的搜索引擎系统(企业级搜索引擎产品)。
2. solr 可以部署到单独的服务器上(WEB 服务)，它可以提供服务，我们的业务系统就只要发送请求，接收响应即可，降低了业务系统的负载。
3. solr 部署在专门的服务器上，它的索引库就不会受业务系统服务器存储空间的限制。
4. solr 支持分布式集群，索引服务的容量和能力可以线性扩展。

6.4.3 工作机制

Solr 索引的实现方法很简单

1. 用 POST 方法向 Solr 服务器发送一个描述 Field 及其内容的 XML 文档。

2. Solr 根据 xml 文档添加、删除、更新索引。

3. Solr 搜索只需要发送 HTTP GET 请求，然后对 Solr 返回 Xml、Json 等格式的查询结果进行解析，组织页面布局。

4. PS：Solr 不提供构建 UI 的功能，Solr 提供了一个管理界面，通过管理界面可以查询 Solr 的配置和运行情况。

6.5 Ik 分词器

6.5.1 什么是分词

把一段中文或者别的划分成一个个的关键字。搜索时候会将用户的搜索关键字进行分词，会把数据库中或者索引库中的数据进行分词，然后对两者进行——匹配操作。

6.5.2 为什么要使用 Ik 分词器？

默认的中文分词会将每个字看成一个词，比如"中国的花"会被分为"中"，"国"，"的"，"花"，这显然是不符合要求的，所以我们需要安装中文分词器 ik 来解决这个问题。

6.5.3 Ik 分词器的原理？

Ik 分词器提供了两个分词算法 ik_smart 和 ik_max_word。其中 ik_smart 为最少切分，ik_max_word 为最细粒度划分。

ik_smart

(1) 最小切分：在浏览器地址栏输入地址
`http://127.0.0.1:9200/_analyze?analyzer=ik_smart&pretty=true&text=我是程序员`
输出的结果为：

```
{
  "tokens" : [
    {
      "token" : "我",
      "start_offset" : 0,
      "end_offset" : 1,
      "type" : "CN_CHAR",
      "position" : 0
    },
    {
      "token" : "是",
      "start_offset" : 1,
      "end_offset" : 2,
      "type" : "CN_CHAR",
      "position" : 1
    },
    {
      "token" : "程序员",
      "start_offset" : 2,
      "end_offset" : 5,
      "type" : "CN_WORD",
      "position" : 2
    }
  ]
}
```

ik_max_word

(2) 最细切分：在浏览器地址栏输入地址
`http://127.0.0.1:9200/_analyze?analyzer=ik_max_word&pretty=true&text=我是程序员`
输出的结果为：

```
{
  "tokens" : [
    {
      "token" : "我",
      "start_offset" : 0,
      "end_offset" : 1,
      "type" : "CN_CHAR",
      "position" : 0
    },
    {
      "token" : "是",
      "start_offset" : 1,
      "end_offset" : 2,
      "type" : "CN_CHAR",
      "position" : 1
    },
    {
      "token" : "程序员",
      "start_offset" : 2,
      "end_offset" : 5,
      "type" : "CN_WORD",
      "position" : 2
    },
    {
      "token" : "程序",
      "start_offset" : 2,
      "end_offset" : 4,
      "type" : "CN_WORD",
      "position" : 3
    },
    {
      "token" : "员",
      "start_offset" : 4,
      "end_offset" : 5,
      "type" : "CN_CHAR",
      "position" : 4
    }
  ]
}
```

6.5.4 Elasticsearch 集成 Ik 分词器

先将其解压，将解压后的文件夹重命名为 ik。

将 ik 文件夹拷贝到 elasticsearch/plugins 目录下。

重新启动，即可加载 Ik 分词器。

6.6 Elasticsearch 和 Solr 对比区别是什么？

1. es 基本是开箱即用，非常简单。而 Solr 安装略微复杂一些。
2. es 自身带有分布式协调管理功能。而 Solr 需利用 Zookeeper 进行分布式管理。
3. Elasticsearch 只支持 json 文件格式。而 Solr 支持更多格式的数据，比如 JSON、XML、CSV。
4. es 本身更侧重于核心功能，高级功能多由第三方插件提供，例如图形化界面需要 kibana 友好支持。而 Solr 官方提供的功能更多。
5. es 建立索引快（即查询慢），即实时性查询快，用于 facebook 新浪等搜索。Solr 查询快，但更新索引时慢（即插入删除慢），用于电商等查询多的应用。Solr 是传统搜索应用的有力解决方案，但 Elasticsearch 更适用于新兴的实时搜索应用。
6. es 相对开发维护者较少，更新太快，学习使用成本较高。Solr 比较成熟，有一个更大，更成熟的用户、开发和贡献者社区。

7. 页面静态化

7.1 业务思路

在静态页面微服务中提供了生产静态页面和删除静态页面的功能。当监听到生产静态页的消息时，监听器会根据要生产静态页面的商品 ID 使用 Feign 组件调用商品微服务用以获取该商品的详细信息，组织成事先协商好的格式数据，调用 Thymeleaf 生产商品的静态页面并保存到 Nginx 静态资源文件夹下。当监听到删除静态页面的消息时，监听器会根据商品 ID 直接删除 Nginx 静态资源文件夹下的对应的静态页面。

7.2 什么是静态化

静态化是指把动态生成的 HTML 页面变为静态内容保存，以后用户的请求到来，直接访问静态页面，不再经过服务的渲染。

而静态的 HTML 页面可以部署在 nginx 中，从而大大提高并发能力，减小 tomcat 压力。

7.3 Thymeleaf

概念

是现代化服务器端的 Java 模板引擎，不同与 JSP 和 FreeMarker，Thymeleaf 的语法更加接近 HTML，并且也有不错的扩展性。是 SpringBoot 官方所推荐使用的。

优势

模板即原型，前后端分离。

8. 用户注册

8.1 业务思路

用户搜索到自己心仪的商品后可能会发生购买，但在购买前必须先登录，所以在此之前需要先注册用户，在用户注册时需要先进行数据校验和获取验证码。

当用户填写完用户名或手机号后，会向后台发送异步请求，后台获取用户名或手机号码会对数据进行唯一校验，若是未注册过就会返回可用，反之返回不可用。

正确填写手机号码后可以点击发送验证码，后台接收到页面发送过来手机号，确认无误后，随机生成 6 位验证码，保存到 Redis 中 5 分钟。调用短信微服务将手机号和验证码发送过去，短信微服务监听到发送验证码消息后，会使用阿里大于发送验证码到指定手机号码。

当用户填写完所有个人信息并且数据校验无误，验证码填写完成后可以进行最后的用户注册。

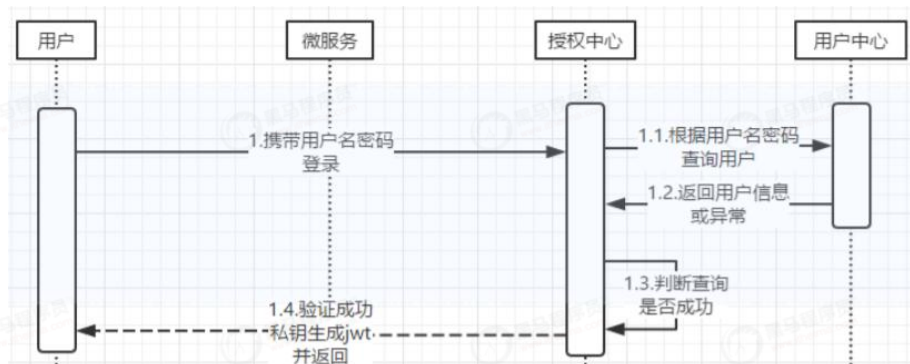
8.2 阿里大于

阿里大于（原阿里大鱼）是阿里通信旗下产品，融合了三大运营商的通信能力，通过传统通信业务和能力与互联网相结合，创新融合阿里巴巴生态内容，全力为中小企业和开发者提供优质服务。阿里大于提供包括短信、语音、流量直充、私密专线、店铺手机号等个性化服务。通过阿里大于，用淘宝帐户打通三大运营商通信能力，通过这个平台，中小企业及开发者可以在最短的时间内实现短信验证码发送、短信服务提醒、语音验证码、语音服务通知、IVR 及呼叫中心、

码号、后向流量、隐私保护相关的能力，实现互联网电信化。

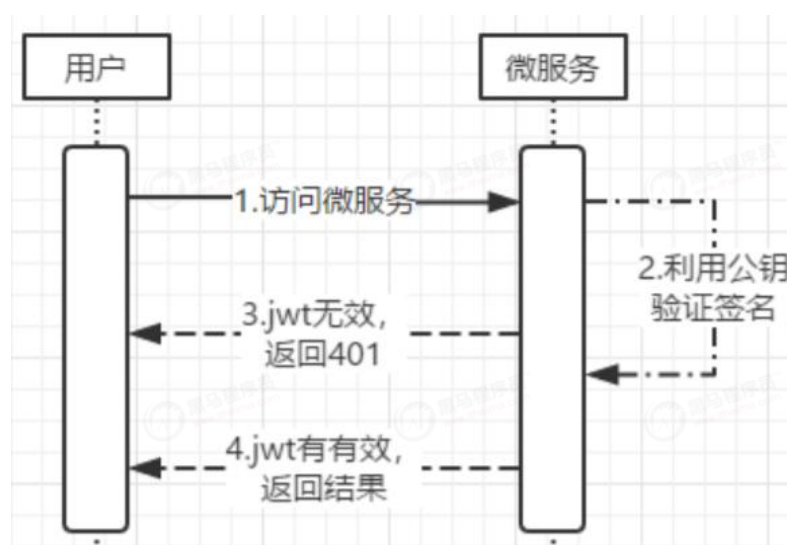
9. 用户登录

9.1 授权业务思路（登录）



用户在登录页面正确填写用户名和密码后，点击登录，请求会携带用户名和密码到授权中心微服务，授权中心微服务使用 Feign 组件访问用户微服务根据用户名和密码查询用户信息，判断返回的用户是否为 NULL，不为 NULL 根据私钥生产 JWT 并写入 cookie 到浏览器。

9.2 鉴权业务思路（每个微服务验证是否登录）



登录成功后用户每次访问微服务都会获取 cookie 中的 JWT，利用公钥验证

token。若是 JWT 无效就会返回 401，有效就会返回用户目标资源。

9.3 什么是有状态登录？

有状态服务，即服务端需要记录每次会话的客户端信息，从而识别客户端身份，根据用户身份进行请求的处理，典型的设计如 tomcat 中的 session。

缺点

服务端保存大量数据，增加服务端压力。

服务端保存用户状态，无法进行水平扩展。

客户端请求依赖服务端，多次请求必须访问同一台服务器。

9.4 什么是无状态登录？

微服务集群中的每个服务，对外提供的都是 Rest 风格的接口。而 Rest 风格的一个最重要的规范就是：服务的无状态性，即：

服务端不保存任何客户端请求者信息。

客户端的每次请求必须具备自描述信息，通过这些信息识别客户端身份。

优点

客户端请求不依赖服务端的信息，任何多次请求不需要必须访问到同一台服务。

服务端的集群和状态对客户端透明。

服务端可以任意的迁移和伸缩。

减小服务端存储压力。

9.5 JWT

JSON Web Token (JWT) 是一个非常轻巧的规范。这个规范允许我们使用 JWT 在用户和服务器之间传递安全可靠的信息。

组成

一个 JWT 实际上就是一个字符串，它由三部分组成，头部、载荷与签名。

头部 (Header)

头部用于描述关于该 JWT 的最基本的信息，例如其类型以及签名所用的算法等。这也可以被表示成一个 JSON 对象。

```
{"typ":"JWT","alg":"HS256"}
```

在头部指明了签名算法是 HS256 算法。 我们进行 BASE64 编码 (<http://base64.xpcha.com/>) ， 编码后的字符串如下：
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

载荷 (payload)

载荷就是存放有效信息的地方。这个名字像是特指飞机上承载的货品，这些有效信息包含三个部分：

1. 标准中注册的声明 (建议但不强制使用)

iss: jwt 签发者

sub: jwt 所面向的用户

aud: 接收 jwt 的一方

exp: jwt 的过期时间，这个过期时间必须要大于签发时间

nbf: 定义在什么时间之前，该 jwt 都是不可用的。

iat: jwt 的签发时间

jwt 的唯一身份标识，主要用来作为一次性 token

2. 公共的声明

公共的声明可以添加任何的信息，一般添加用户的相关信息或其他业务需要的必要信息。但不建议添加敏感信息，因为该部分在客户端可解密。

3. 私有的声明

私有声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为 base64 是对称解密的，意味着该部分信息可以归类为明文信息。

这个指的就是自定义的 claim。比如前面那个结构举例中的 admin 和 name 都属于自定义的 claim。这些 claim 跟 JWT 标准规定的 claim 区别在于：

JWT 规定的 claim，JWT 的接收方在拿到 JWT 之后，都知道怎么对这些标准的 claim 进行验证(还不知道是否能够验证)；而 private claims 不会验证，除非明确告诉接收方要对这些 claim 进行验证以及规则才行。定义一个 payload {"sub":"1234567890","name":"John Doe","admin":true}

然后将其进行 base64 加密，得到 Jwt 的第二部分。

```
eyJzdWiiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjYWRtaW4iOnRydWV9
```

签名 (signature)

jwt 的第三部分是一个签名信息，这个签名信息由三部分组成：

header (base64 后的)

payload (base64 后的)

secret

这个部分需要 base64 加密后的 header 和 base64 加密后的 payload 使用。连接组成的字符串，然后通过 header 中声明的加密方式进行加盐 secret 组合加密，然后就构成了 jwt 的第三部分。

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

注意

secret 是保存在服务器端的，jwt 的签发生成也是在服务器端的，secret 就是用来进行 jwt 的签发和 jwt 的验证，所以，它就是你服务端的私钥，在任何场景都不应该流露出去。一旦客户端得知这个 secret，那就意味着客户端是可以自我签发 jwt 了。

使用场景

1. 一次性验证

比如用户注册后需要发一封邮件让其激活账户，通常邮件中需要有一个链接，这个链接需要具备以下的特性：能够标识用户，该链接具有时效性（通常只允许几小时之内激活），不能被篡改以激活其它可能的账户.....这种场景就和 jwt 的特性非常贴近，jwt 的 payload 中固定的参数：iss 签发者和 exp 过期时间正是为其做准备的。

2. restful api 的无状态认证

使用 jwt 来做 restful api 的身份认证也是值得推崇的一种使用方案。客户端和服务端共享 secret；过期时间由服务端校验，客户端定时刷新；签名信息不可被修改.....spring security oauth jwt 提供了一套完整的 jwt 认证体系，

以笔者的经验来看：使用 oauth2 或 jwt 来做 restful api 的认证都没有大问题，oauth2 功能更多，支持的场景更丰富，后者实现简单。

3.使用 jwt 做单点登录+会话管理(不推荐)

JWT token 泄露了怎么办？

使用 https 加密你的应用，返回 jwt 给客户端时设置 httpOnly=true 并且使用 cookie 而不是 LocalStorage 存储 jwt，这样可以防止 XSS 攻击和 CSRF 攻击。

Secret 如何设计？

jwt 唯一存储在服务端的只有一个 secret，个人认为这个 secret 应该设计成和用户相关的属性，而不是一个所有用户公用的统一值。这样可以有效的避免一些注销和修改密码时遇到的窘境。

注销和修改密码？

传统的 session+cookie 方案用户点击注销，服务端清空 session 即可，因为状态保存在服务端。但 jwt 的方案就比较难办了，因为 jwt 是无状态的，服务端通过计算来校验有效性。没有存储起来，所以即使客户端删除了 jwt，但是该 jwt 还是在有效期内，只不过处于一个游离状态。分析下痛点：注销变得复杂的原因在于 jwt 的无状态。提供几个方案，视具体的业务来决定能不能接受：

仅仅清空客户端的 cookie，这样用户访问时就不会携带 jwt，服务端就认为用户需要重新登录。这是一个典型的假注销，对于用户表现出退出的行为，实际上这个时候携带对应的 jwt 依旧可以访问系统。

清空或修改服务端的用户对应的 secret，这样在用户注销后 jwt 本身不变，

但是由于 secret 不存在或改变，则无法完成校验。这也是为什么将 secret 设计成和用户相关的原因。

借助第三方存储自己管理 jwt 的状态，可以以 jwt 为 key，实现去 Redis 一类的缓存中间件中去校验存在性。方案设计并不难，但是引入 Redis 之后，就把无状态的 jwt 硬生生变成了有状态了，违背了 jwt 的初衷。实际上这个方案和 session 都差不多了。

修改密码则略微有些不同，假设号被到了，修改密码（是用户密码，不是 jwt 的 secret）之后，盗号者在原 jwt 有效期之内依旧可以继续访问系统，所以仅仅清空 cookie 自然是不够的，这时，需要强制性的修改 secret。

续签问题

传统的 cookie 续签方案一般都是框架自带的，session 有效期 30 分钟，30 分钟内如果有访问，session 有效期被刷新至 30 分钟。而 jwt 本身的 payload 之中也有一个 exp 过期时间参数，来代表一个 jwt 的时效性，而 jwt 想延期这个 exp 就有点身不由己了，因为 payload 是参与签名的，一旦过期时间被修改，整个 jwt 串就变了，jwt 的特性天然不支持续签。

解决方案

1. 每次请求刷新 jwt。

jwt 修改 payload 中的 exp 后整个 jwt 串就会发生改变，那就让它变好了，每次请求都返回一个新的 jwt 给客户端。只是这种方案太暴力了，会带来性能问题。

2. 只要快要过期的时候刷新 jwt

此方案是基于上个方案的改造版，只在前一个 jwt 的最后几分钟返回给客户

端一个新的 jwt。这样做，触发刷新 jwt 基本就要看运气了，如果用户恰巧在最后几分钟访问了服务器，触发了刷新，万事大吉。如果用户连续操作了 27 分钟，只有最后的 3 分钟没有操作，导致未刷新 jwt，无疑会令用户抓狂。

3. 完善 refreshToken

借鉴 oauth2 的设计，返回给客户端一个 refreshToken，允许客户端主动刷新 jwt。一般而言，jwt 的过期时间可以设置为数小时，而 refreshToken 的过期时间设置为数天。

4. 使用 Redis 记录独立的过期时间

在 Redis 中单独为每个 jwt 设置了过期时间，每次访问时刷新 jwt 的过期时间，若 jwt 不存在与 Redis 中则认为过期。

9.6 加密技术

1. 加密技术的类型

2. 加密技术是对信息进行编码和解码的技术，编码是把原来可读信息（又称明文）译成代码形式（又称密文），其逆过程就是解码（解密），加密技术的要点是加密算法，加密算法可以分为三类：

对称加密，如 AES

基本原理：将明文分成 N 个组，然后使用密钥对各个组进行加密，形成各自的密文，最后把所有的分组密文进行合并，形成最终的密文。

优势：算法公开、计算量小、加密速度快、加密效率高

缺陷：双方都使用同样密钥，安全性得不到保证

非对称加密，如 RSA

基本原理：同时生成两把密钥：私钥和公钥，私钥隐秘保存，公钥可以下发给信任客户端私钥加密，持有私钥或公钥才可以解密公钥加密，持有私钥才可解密

优点：安全，难以破解

缺点：算法比较耗时

不可逆加密，如 MD5，SHA

基本原理：加密过程中不需要使用密钥，输入明文后由系统直接经过加密算法处理成密文，这种加密后的数据是无法被解密的，无法根据密文推算出明文。

RSA

1977 年，三位数学家 Rivest、Shamir 和 Adleman 设计的一种算法，可以实现非对称加密。这种算法用他们三个人的名字缩写：RSA。

为什么要使用 RSA 非对称加密

使用非对称加密的特性，不用担心公钥泄漏问题，因为公钥是无法伪造签名的。

10. 购物车

10.1 业务思路

考虑到用户未登录和已登录情况下都可能发生将商品加入购物车的行为，所以我们的购物车也分为未登录下的购物车和已登录下的购物车。

在用户未登录的情况下，如果将商品加入到购物车，会先到 localStorage 中查询是否存在相同的商品，如果有，那么数量相加，如果没有，将商品数据加入到 localStorage 中。

在用户已登录的情况下，如果将商品加入到购物车，会先到 Redis 中查询此

用户的购物车中是否存在相同的商品，如果有，那么数量相加，如果没有，将商品数据加入到 Redis 的此用户的购物车中。

当用户登录后，查看购物车时会在查询购物车列表前检查用户的 localStorage。若是 localStorage 中有商品就提交到后台，整合到 Redis 中，清空 localStorage 中的数据。

10.2 Html5 web 存储

Html5 提供了两种在客户端存储数据的新方法：

localStorage：没有时间限制的数据存储

SessionStorage：针对一个 session 的数据存储（关闭浏览器）

之前这些都是由 cookie 完成的。但是 cookie 不适合大量数据的存储，因为它们由每个对服务器的请求来传递，这使得 cookie 速度很慢而且效率也不高。

在 Html5 中，数据不是由每个服务器请求传递的，而是只有在请求时使用数据。它使在不影响网站性能的情况下存储大量数据成为可能。

对于不同的网站，数据存储于不同的区域，并且一个网站只能访问其自身的数据。

Html5 使用 JavaScript 来存储和访问数据。

注意：localStorage 和 SessionStorage 都只能保存字符串。

10.3 Redis

10.3.1 什么是 Redis

是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的

日志型、高性能的 Key-Value 数据库，并提供多种语言的 API。

10.3.2 Redis 的存储结构有哪些

String，字符串，是 Redis 的最基本的类型，一个 key 对应一个 value。

是二进制安全的，最大能存储 512MB。

Hash，散列，是一个键值(key=>value)对集合。string 类型的 field 和 value 的映射表，特别适合用于存储对象。每个 hash 可以存储 $2^{32}-1$ 键值对（40 多亿）

List，列表，是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。最多可存储 $2^{32}-1$ 元素（4294967295，每个列表可存储 40 多亿）。

Set，集合，是 string 类型的无序集合，最大的成员数为 $2^{32}-1$ （4294967295，每个集合可存储 40 多亿个成员）。

Sorted set，有序集合，和 set 一样也是 string 类型元素的集合，且不允许重复的成员。不同的是每个元素都会关联一个 double 类型的分数。Redis 正是通过分数来为集合中的成员进行从小到大的排序。zset 的成员是唯一的，但分数 (score) 却可以重复。

10.3.3 Redis 的优点

1. 纯内存操作。
2. 单线程操作，避免了频繁的上下文切换。
3. 采用了非阻塞 I/O 多路复用机制。

I/O 多路复用机制

I/O 多路复用就是只有单个线程，通过跟踪每个 I/O 流的状态，来管理多个 I/O 流。

10.3.4 Redis 的缺点

缓存和数据库双写一致性问题

一致性的问题很常见，因为加入了缓存之后，请求是先从 Redis 中查询，如果 Redis 中存在数据就不会走数据库了，如果不能保证缓存跟数据库的一致性就会导致请求获取到的数据不是最新的数据。

解决方案：

1、编写删除缓存的接口，在更新数据库的同时，调用删除缓存的接口删除缓存中的数据。这么做会有耦合高以及调用接口失败的情况。

2、消息队列：ActiveMQ，消息通知。

缓存的并发竞争问题

并发竞争，指的是同时有多个子系统去 set 同一个 key 值。

解决方案：最简单的方式就是准备一个分布式锁，大家去抢锁，抢到锁就做 set 操作即可

缓存雪崩问题

缓存雪崩，即缓存同一时间大面积的失效，这个时候又来了一波请求，结果请求都怼到数据库上，从而导致数据库连接异常。

解决方案：

1. 给缓存的失效时间，加上一个随机值，避免集体失效。
2. 使用互斥锁，但是该方案吞吐量明显下降了。

3. 搭建 Redis 集群。

缓存击穿问题

缓存穿透，即黑客故意去请求缓存中不存在的数据，导致所有的请求都打到数据库上，从而数据库连接异常。

解决方案：

1、利用互斥锁，缓存失效的时候，先去获得锁，得到锁了，再去请求数据库。没得到锁，则休眠一段时间重试

2、采用异步更新策略，无论 key 是否取到值，都直接返回，value 值中维护一个缓存失效时间，缓存如果过期，异步起一个线程去读数据库，更新缓存。

10.3.5 Redis 的持久化

Redis 提供了两种持久化的方式，分别是 RDB(Redis DataBase)和 AOF(Append Only File)。

RDB，简而言之，就是在不同的时间点，将 Redis 存储的数据生成快照并存储到磁盘等介质上。

AOF，则是换了一个角度来实现持久化，那就是将 Redis 执行过的所有写指令记录下来，在下次 Redis 重新启动时，只要把这些写指令从前到后再重复执行一遍，就可以实现数据恢复了。

RDB 和 AOF 两种方式也可以同时使用，在这种情况下，如果 Redis 重启的话，则会优先采用 AOF 方式来进行数据恢复，这是因为 AOF 方式的数据恢复完整度更高。

10.3.6 RDB

RDB 持久化方式,是将 Redis 某一时刻的数据持久化到磁盘中,是一种快照式的持久化方法。

RDB 持久化方式:Redis 在进行数据持久化的过程中,会先将数据写入到一个临时文件中,待持久化过程都结束了,才会用这个临时文件替换上次持久化好的文件。正是这种特性,让我们可以随时来进行备份,因为快照文件总是完整可用的。

对于 RDB 方式,Redis 会单独创建一个子进程来进行持久化,而主进程是不会进行任何 IO 操作的,这样就确保了 Redis 极高的性能。

RDB 优点: 如果需要进行大规模数据的恢复,且对于数据恢复的完整性不是非常敏感,那 RDB 方式要比 AOF 方式更加的高效。

RDB 缺点: 如果你对数据的完整性非常敏感,那么 RDB 方式就不太适合你,因为即使你每 5 分钟都持久化一次,当 Redis 故障时,仍然会有近 5 分钟的数据丢失。所以,Redis 还提供了另一种持久化方式,那就是 AOF。

10.3.7 AOF

AOF 方式是将执行过的写指令记录下来,在数据恢复时按照从前到后的顺序再将指令都执行一遍。

实现方式 我们通过配置 Redis.conf 中的 appendonly yes 就可以打开 AOF 功能。如果有写操作(如 SET 等),Redis 就会被追加到 AOF 文件的末尾。

AOF 持久化的方式:默认的 AOF 持久化策略是每秒钟 fsync 一次(fsyntax 是指把缓存中的写指令记录到磁盘中),因为在这种情况下,Redis 仍然可以保

持很好的处理性能，即使 Redis 故障，也只会丢失最近 1 秒钟的数据。

如果在追加日志时，恰好遇到磁盘空间满或断电等情况导致日志写入不完整，也没有关系，Redis 提供了 Redis-check-aof 工具，可以用来进行日志修复。

因为采用了追加方式，如果不做任何处理的话，AOF 文件会变得越来越大，为此，Redis 提供了 AOF 文件重写（rewrite）机制，即当 AOF 文件的大小超过所设定的阈值时，Redis 就会启动 AOF 文件的内容压缩，只保留可以恢复数据的最小指令集。举个例子或许更形象，假如我们调用了 100 次 INCR 指令，在 AOF 文件中就要存储 100 条指令，但这明显是很低效的，完全可以把这 100 条指令合并成一条 SET 指令，这就是重写机制的原理。

AOF 优点：我们通过一个场景再现来说明。某同学在操作 Redis 时，不小心执行了 FLUSHALL，导致 Redis 内存中的数据全部被清空了，这是很悲剧的事情。不过这也不是世界末日，只要 Redis 配置了 AOF 持久化方式，且 AOF 文件还没有被重写（rewrite），我们就可以用最快的速度暂停 Redis 并编辑 AOF 文件，将最后一行的 FLUSHALL 命令删除，然后重启 Redis，就可以恢复 Redis 的所有数据到 FLUSHALL 之前的状态了。是不是很神奇，这就是 AOF 持久化方式的好处之一。但是如果 AOF 文件已经被重写了，那就无法通过这种方法来恢复数据了。

AOF 缺点：比如在同样数据规模的情况下，AOF 文件要比 RDB 文件的体积大。而且，AOF 方式的恢复速度也要慢于 RDB 方式。

如果你直接执行 BGREWRITEAOF 命令，那么 Redis 会生成一个全新的 AOF 文件，其中便包括了可以恢复现有数据的最少的命令集。

如果运气比较差，AOF 文件出现了被写坏的情况，也不必过分担忧，Redis

并不会贸然加载这个有问题的 AOF 文件，而是报错退出。这时可以通过以下步骤来修复出错的文件：

1. 备份被写坏的 AOF 文件。
2. 运行 `Redis-check-aof -fix` 进行修复。
3. 用 `diff -u` 来看下两个文件的差异，确认问题点。
4. 重启 Redis，加载修复后的 AOF 文件。

10.3.8 Redis 集群

从复制

主从复制原理

从服务器连接主服务器，发送 SYNC 命令。

主服务器接收到 SYNC 命令后，开始执行 BGSAVE 命令生成 RDB 文件并使用缓冲区记录此后执行的所有写命令。

主服务器 BGSAVE 执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令。

从服务器收到快照文件后丢弃所有旧数据，载入收到的快照。

主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令。

从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令（从服务器初始化完成）。

主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令（从服务器初始化完成后的操作）。

优点

支持主从复制，主机会自动将数据同步到从机，可以进行读写分离。

为了分载 Master 的读操作压力，Slave 服务器可以为客户端提供只读操作的服务，写服务仍然必须由 Master 来完成。

Slave 同样可以接受其它 Slaves 的连接和同步请求，这样可以有效的分载 Master 的同步压力。

Master Server 是以非阻塞的方式为 Slaves 提供服务。所以在 Master-Slave 同步期间，客户端仍然可以提交查询或修改请求。

Slave Server 同样是以非阻塞的方式完成数据同步。在同步期间，如果有客户端提交查询请求，Redis 则返回同步之前的数据。

缺点

Redis 不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的 IP 才能恢复。

主机宕机，宕机前有部分数据未能及时同步到从机，切换 IP 后还会引入数据不一致的问题，降低了系统的可用性。

Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。

哨兵模式

当主服务器中断服务后，可以将一个从服务器升级为主服务器，以便继续提供服务，但是这个过程需要人工手动来操作。为此，Redis2.8 中提供了哨兵工具来实现自动化的系统监控和故障恢复功能。

哨兵的作用就是监控 Redis 系统的运行状况，它的功能包括以下两个。

- 1、监控主服务器和从服务器是否正常运行。
- 2、主服务器出现故障时自动将从服务器转换为主服务器。

哨兵的工作方式

每个 Sentinel (哨兵) 进程以每秒钟一次的频率向整个集群中的 Master 主服务器, Slave 从服务器以及其他 Sentinel (哨兵) 进程发送一个 PING 命令。

如果一个实例 (instance) 距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值, 则这个实例会被 Sentinel (哨兵) 进程标记为主观下线 (SDOWN)。

如果一个 Master 主服务器被标记为主观下线 (SDOWN), 则正在监视这个 Master 主服务器的所有 Sentinel (哨兵) 进程要以每秒一次的频率确认 Master 主服务器的确进入了主观下线状态。

当有足够数量的 Sentinel (哨兵) 进程 (大于等于配置文件指定的值) 在指定的时间范围内确认 Master 主服务器进入了主观下线状态 (SDOWN), 则 Master 主服务器会被标记为客观下线 (ODOWN)。

在一般情况下, 每个 Sentinel (哨兵) 进程会以每 10 秒一次的频率向集群中的所有 Master 主服务器、Slave 从服务器发送 INFO 命令。

当 Master 主服务器被 Sentinel (哨兵) 进程标记为客观下线 (ODOWN) 时, Sentinel (哨兵) 进程向下线的 Master 主服务器的所有 Slave 从服务器发送 INFO 命令的频率会从 10 秒一次改为每秒一次。

若没有足够数量的 Sentinel (哨兵) 进程同意 Master 主服务器下线, Master 主服务器的客观下线状态就会被移除。若 Master 主服务器重新向 Sentinel (哨兵) 进程发送 PING 命令返回有效回复, Master 主服务器的主观下线状态就会被移除。

优点

哨兵模式是基于主从模式的，所有主从的优点，哨兵模式都具有。

主从可以自动切换，系统更健壮，可用性更高。

缺点

Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。

Redis-Cluster 集群

Redis 的哨兵模式基本已经可以实现高可用，读写分离，但是在这种模式下每台 Redis 服务器都存储相同的数据，很浪费内存，所以在 Redis3.0 上加入了 cluster 模式，实现的 Redis 的分布式存储，也就是说每台 Redis 节点上存储不同的内容。

Redis-Cluster 采用无中心结构,它的特点如下：

所有的 Redis 节点彼此互联(PING-PONG 机制),内部使用二进制协议优化传输速度和带宽。

节点的 fail 是通过集群中超过半数的节点检测失效时才生效。

客户端与 Redis 节点直连,不需要中间代理层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可。

工作方式

在 Redis 的每一个节点上，都有这么两个东西，一个是插槽 (slot)，它的取值范围是：0-16383。还有一个就是 cluster，可以理解为是一个集群管理的插件。当我们的存取的 key 到达的时候，Redis 会根据 crc16 的算法得出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，通过这个值，去找到对应的插槽所对应的节点，然后直接自动跳转到这个对应的节点上进行存取操作。

为了保证高可用，Redis-cluster 集群引入了主从模式，一个主节点对应一个或者多个从节点，当主节点宕机的时候，就会启用从节点。当其它主节点 ping 一个主节点 A 时，如果半数以上的主节点与 A 通信超时，那么认为主节点 A 宕机了。如果主节点 A 和它的从节点 A1 都宕机了，那么该集群就无法再提供服务了。

10.4 购物车存 cookie 里边 可以实现不登录就可以使用购物车 那么现在没有登录把商品存购物车了 登录了 然后换台电脑并且登录了还能不能看见购物车的信息？如果看不到怎么做到 cookie 同步，就是在另外一台电脑上可以看到购物车信息

乐优商城现阶段使用的仅仅是把购物车的商品写入 cookie 中，这样服务端基本上没有存储的压力。但是弊端就是用户更换电脑后购物车不能同步。

注意：以下这部分是淘淘商城的回答

我们的乐优商城项目是把购物车信息保存在了客户端 localStorage 本地存储,未登录时可以查询到购物车(通过 SpuID 查询所有 Sku)

打算下一步这么实现 当用户没有登录时向购物车添加商品是添加到 cookie 中,当用户登录后购物车的信息是存储在 Redis 中的并且是跟用户 id 向关联的，此时你更换电脑后使用同一账号登录购物车的信息就会展示出来。

10.5 如果用户一直添加购物车添加商品怎么办？并且他添加一次你查询一次数据库？互联网上用户那么多，这样会对数据库造成很大压力怎么办？

当前我们使用 cookie 的方式来保存购物车的数据，所以当用户往购物车中添加商品时，并不对数据库进行操作。将来把购物车商品放入 Redis 中，Redis 是可以持久化的可以永久保存，此时就算是频繁的往购物车中添加数据也没什么问题。

10.6 Redis 为什么可以做缓存？项目中使用 Redis 的目的是什么？Redis 什么时候使用？

1.Redis 是 key-value 形式的 nosql 数据库。可以快速的定位到所查找的 key，并把其中的 value 取出来。并且 Redis 的所有数据都是放到内存中，存取的速度非常快，一般都是用来做缓存使用。

2.项目中使用 Redis 一般都是作为缓存来使用的，缓存的目的就是为了减轻数据库的压力提高存取效率。

3.在互联网项目中只要是涉及高并发或者是存在大量读数据的情况下都可以使用 Redis 作为缓存。当然 Redis 提供丰富的数据类型，除了缓存还可以根据实际的业务场景来决定 Redis 的作用。例如使用 Redis 保存用户的购物车信息、生成订单号、访问量计数器、任务队列、排行榜等。

10.7 对 Redis 和 Memcache 有没有了解,为什么选择 Redis?

Memcache 无持久化: Memecache 把数据全部存在内存之中,断电后会挂掉且数据不能超过内存大小。而 Redis 有部份存在硬盘上,这样能保证数据的持久性(默认 RDB 模式)

Memcache 只支持 K-V 结构,Redis 有复杂的数据类型。

Memcache 是多线程,性能比 Redis(单线程)差

在这里可能会有人问:多线程不是应该比单线程性能要更好吗?

这是因为 Memcache 大多数时间用在了切换线程上。

10.8 Redis 面试经常会问到的问题

在这里列举一下,因为我也还没有去特别了解.后续可能会补上

持久化 避免缓存击穿

分片(集群) 避免缓存雪崩

主从 哨兵 避免缓存热点,key 过期

11. 订单系统

11.1 业务思路

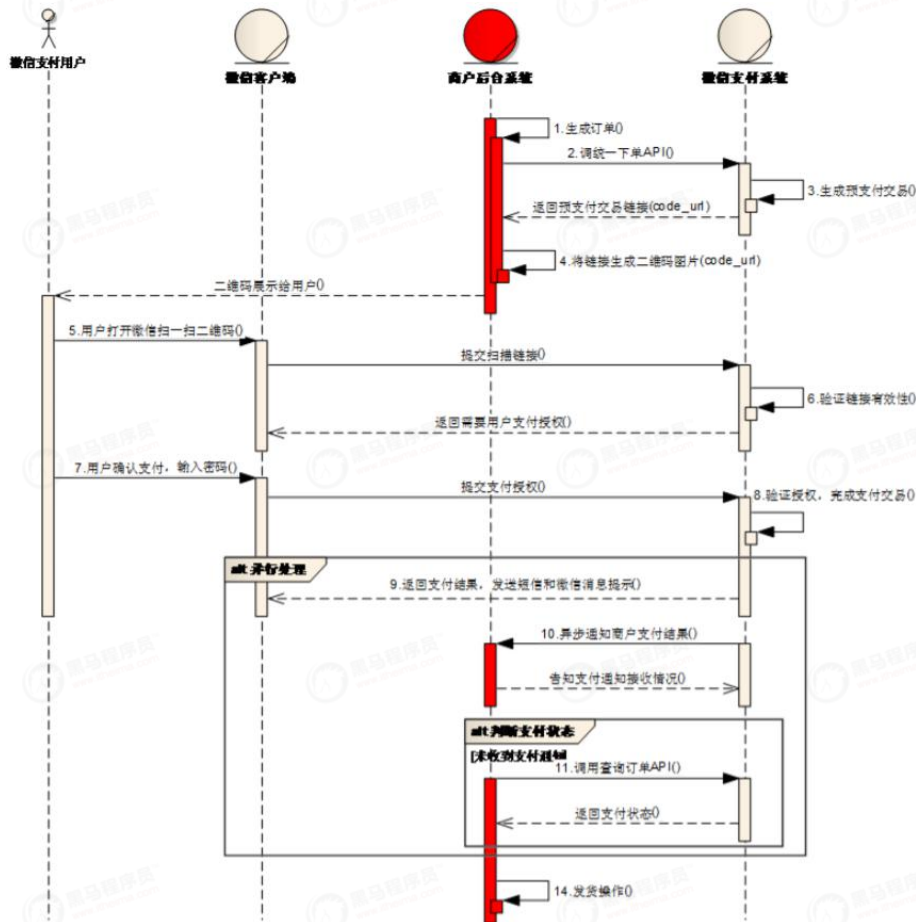
当我们提交订单时,会设置半小时的支付时间,如果这半小时之内为支付的话,我们会将该订单取消,同时将购物车的商品删除。如果半小时之内支付成功的话会生成一个订单信息,订单的状态显示为待发货。我们会根据订单的收件人去调用物流的接口生成物流单,通知物流人员取货。当快递人员提取商品以后,

订单的状态更改为已发货,同时买家可以在订单信息页面的物流状态查看物流信息。物流信息我们采用的定时任务,没两个小时通过物流给的接口中获取物流信息,如果查询出来的物流信息没做改变的话我们不更新数据,如果信息改变我们会在之前的物流信息下增加一个新的物流节点信息。当快递工作人员将商品送到用户手中以后,我们的订单状态会改变成已收货。用户可以在这个订单下评价这个商品。

当用户订单下完以后发现购买有误或者收到商品时发现有问题可以申请退货或者换货的流程,点击申请退货按钮以后我们会根据这笔订单去生成一笔退货单申请,退货单申请由商家负责审核,如果审核通过买家将商品寄回,当商家收到商品以后根据退货单的退货信息根据商品比较确认正确的时候关闭退货单,货款会退回到买家。

12.支付系统

12.1 业务思路(微信支付)



1. 商户系统根据用户选择的商品生成订单。
2. 用户确认支付后根据微信【统一下单 API】，向微信支付系统发出请求（我们通过 httpclient 方式请求的）
3. 微信支付系统收到请求后，先生成预支付订单，然后给商户系统返回二维码连接。
4. 商户系统拿到返回值字符串，转换成对象，然后取出二维码连接生成二维码。
5. 用户通过微信“扫一扫”功能扫描二维码，微信客户端将扫码内容发送到微信支付系统。
6. 微信支付系统接收到支付请求，验证链接有效性后发起支付，请求客户确认，然后我们的微信端就会弹出需要确认支付的页面。
7. 用户输入密码，确认支付并提交授权。
8. 微信支付系统根据用户授权完成交易。
9. 微信支付系统支付成功后向微信客户端返回交易结果，并将交易结果通过短信、微信提

示用户。

10. 微信支付系统通过发送异步消息通知商户系统后台支付结果,商户系统需回复接收情况,通知微信支付系统不再发送该单的通知。
11. 未收到支付通知的情况,商户系统可调用【查询订单 APP】。
12. 商户确认订单已经支付后给用户发货。

12.2 支付接口是怎么做的？

面试中可以说支付这部分不是我们做的,我们项目中并没有涉及支付部分的处理。如果了解支付是如何实现可以参考之前学过的易宝支付相关处理以及支付宝、微信支付相关文档。

12.3 第一个是当两个客户同时买一件商品时库存只有一个了,怎么控制？

可以使用 MySQL 的行锁机制,实现乐观锁,在更新商品之前将商品锁定,其他用户无法读取,当此用户操作完毕后释放锁。当并发量高的情况下,需要使用缓存工具例如 Redis 来管理库存。

十. 十次方项目

1. 十次方背景

1.1.需求分析

首先,该项目工程采用的是前后端分离的开发形式。为什么要采用前后端分

离的开发形式呢？

《十次方》是程序员的专属社交平台，包括头条.问答.活动.交友.吐槽.招聘六大频道。

十次方名称的由来:2 的 10 次方为 1024，程序员都懂的。

1.2 前后端分离的优点和必要性

前端 JS 做大部分的数据处理工作，减少对服务器的压力；

后端错误不会直接反映到前台，提高用户体验度；

不同终端（pad/mobile/pc）的兴起，单一浏览器端的访问已经不能满足用户的需求；

JS 能够很好的适应前端的效果但是很难做到与服务器的通讯，后端又无法很好的去调整前端页面；

提高开发效率，后端负责业务/数据接口，前端负责展现/交互逻辑，同一份数据接口，可以定制开发多个版本。（一般项目后期会针对性的整理开发接口 API 方便内外部的调用访问）

1.3 什么是前后端分离

经由分析讨论以及资料的收集，我们所一致认同的概念是 SPA(single-page application)，所有用到的展现数据都是后端通过异步接口（ajax.json）的方式实现的，前端直管展现。

从某种意义上来说，SPA 确实做到了前后端分离，但这种方式存在两个问题：

(1) WEB 服务中, SPA 类占的比例很少。很多场景下还有同步/同步+异步混合的模式, SPA 不能作为一种通用的解决方案;现阶段的 SPA 开发模式,接口通常是按照展现逻辑来提供的,有时候为了提高效率,后端会帮我们处理一些展现逻辑,这就意味着后端还是涉足了 View 层的工作,不是真正的前后端分离。

(2) SPA 式的前后端分离,是从物理层做区分(认为只要是客户端的就是前端,服务器端的就是后端),这种分法已经无法满足我们前后端分离的需求,我们认为从职责上划分才能满足目前我们的使用场景:

前端:负责 View 和 Controller 层;只负责 Model 层,业务处理/数据等。

1.4.为什么要前后端分离

项目背景:随着互联网的发展,在一定程度上极大的拉近了人与人之间的距离却又在某种程度上增加了人与人之间的隔膜。我们迫切的想要更多人的关注却往往陷入社交平台泛滥的漩涡,基于这种情况我们开发了《十次方》应用于 IT 方面同学的沟通.交流。让同样的人,说同样的话,得到更多的共鸣与理解。

页面原型的介绍.展示:

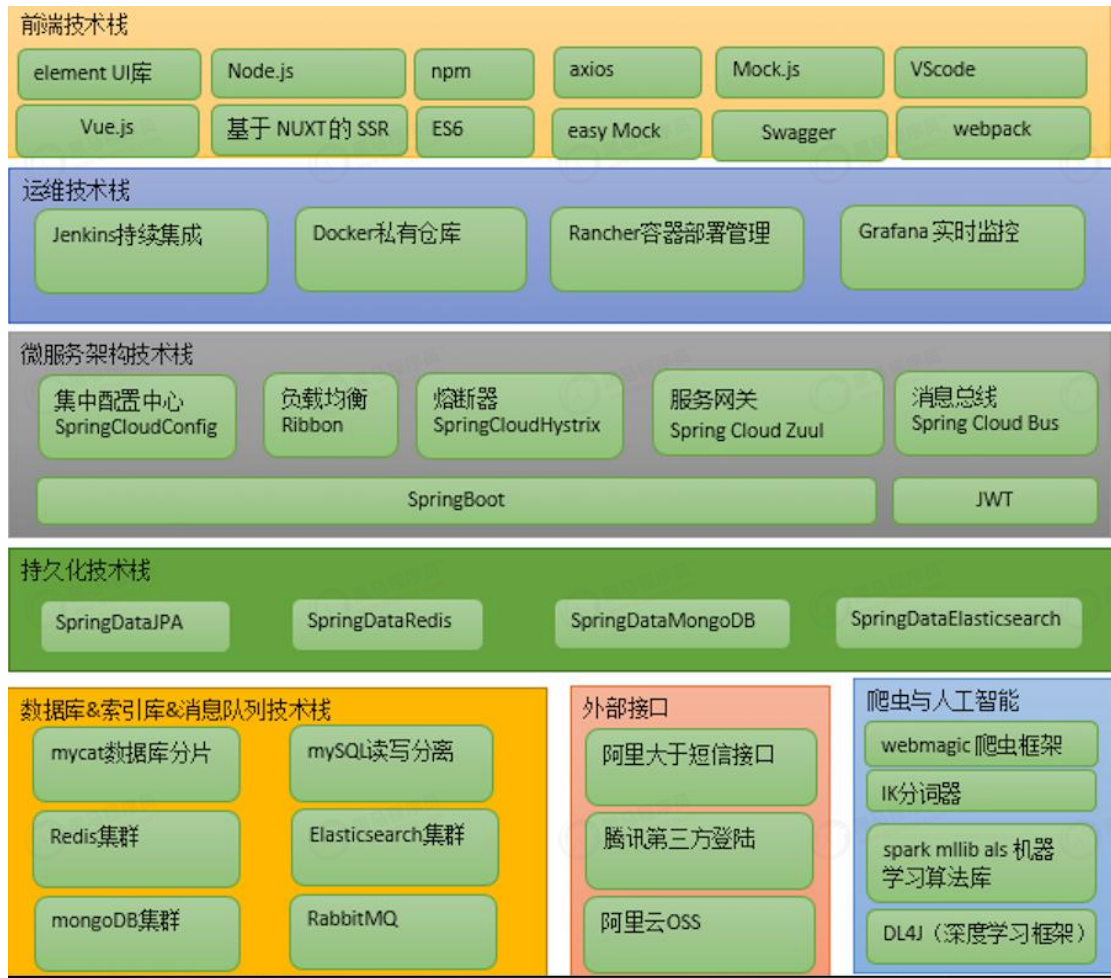


以上，是首页的原型展示。其余界面原型在后面功能实现过程中逐步进行展示。

除此之外，项目开发过程中一般还会有项目需求文档（一般由产品方面来撰写）有的公司可能为了加快开发进度往往先将这部分工作放一放以导致于后期进行项目调整的时候带来诸多屏障。所以，在这一块我还是建议无论项目紧张与否这方面还是要有的。

1.5.十次方项目背景

1.5.1 系统架构：



《十次方》采用前后端分离的系统架构，后端架构为：

SpringBoot+SpringCloud+SpringMVC+SpringData

我们把这种架构也称之为全家桶。

1.5.2 特色

十次方采用了当前主流的前后端分离的开发模式，后端使用 Spring 全家桶框架（即 SpringBoot + SpringCloud + Spring Data + Spring MVC）开发微服务；前端使用以 Node.js 为核心的 Vue 全套生态解决方案。项目中涵盖了微服务认证.微服务网关.微服务熔断.微服务集中配置.微服务持续集成.第三方登陆.云存储.爬虫.人工智能.单页面(SPA).服务端渲染（SSR）等 30 余种解决方案。

采用前后端分离的方式进行系统开发

采用模块化的课程设计，分为微服务开发，前端系统开发，爬虫与人工智能
开发三个模块

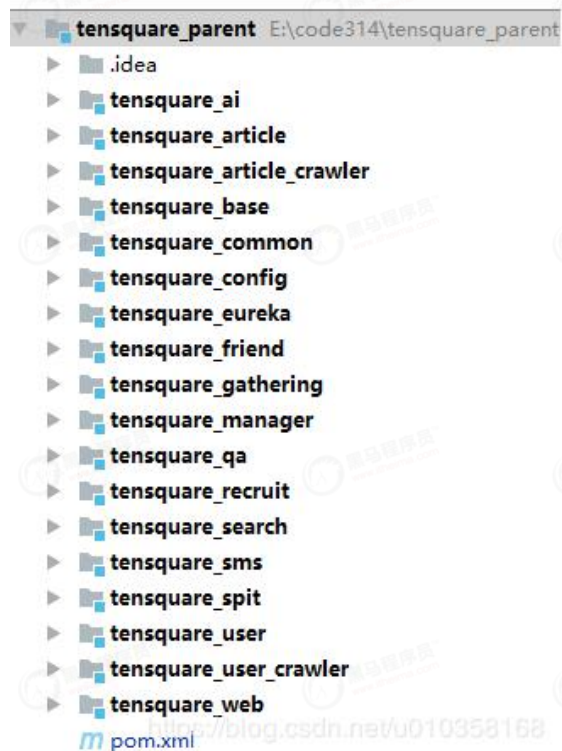
打造 Java 全栈式工程师,让学员站在 Java 软件开发的金字塔顶端



1.5.3 模块划分

十次方工程共分为 18 个子模块(其中 17 个是微服务)。

如下：



模块名称 模块中文名称

tensquare_common 公共模块

tensquare_article 文章微服务

tensquare_base 基础微服务

tensquare_friend 交友微服务

tensquare_gathering 活动微服务

tensquare_qa 问答微服务

tensquare_recruit 招聘微服务

tensquare_user 用户微服务

tensquare_spit 吐槽微服务

tensquare_search 搜索微服务

tensquare_web 前台微服务网关

tensquare_manager	后台微服务网关
tensquare_eureka	注册中心
tensquare_config	配置中心
tensquare_sms	短信微服务
tensquare_article_crawler	文章爬虫微服务
tensquare_user_crawler	用户爬虫微服务
tensquare_ai	人工智能微服务

1.5.4 表结构分析

文件名	数据库说明
tensquare_article.sql	文章
tensquare_base.sql	基础
tensquare_friend.sql	交友
tensquare_gatherinng.sql	活动
tensquare_qa.sql	问答
tensquare_recruit.sql	招聘
tensquare_user.sql	用户

tensquare_article数据库, tb_article表			
文章表	tb_article		
字段名称	字段含义	字段类型	备注
id	ID	文本	
columnid	专栏ID	文本	
userid	用户ID	文本	
title	文章标题	文本	
content	文章内容	文本	
image	文章封面	文本	
createtime	发表日期	日期	
updatetime	修改日期	日期	
ispublic	是否公开	文本	0: 不公开
istop	是否置顶	文本	0: 不置顶
visits	浏览量	整型	
thumbup	点赞数	整型	
comment	评论数	整型	
state	审核状态	文本	0: 未审核 1: 已审核
channelid	所属频道	整型	关联频道表ID
url	URL地址	文本	
type	文章类型	文本	0: 分享

1.5.5 API 文档

前后端约定返回码列表：

状态码	说明
20000	成功
20001	失败
20002	用户名或密码错误
20003	权限不足
20004	远程调用失败
20005	重复操作

1.5.6 开发环境

软件	版本
JDK	1.8
数据库	MySQL 5.7
开发工具	IDEA 2017.1
maven	3.3.9
docker	1.13.1
操作系统	centos 7

1.5.7 项目主要技术以及功能实现

1. 前端环境搭建

章节内容：十次方需求.技术架构，理解前后端分离开发模式, Node.js 基本使用方法，理解模块化编程, 包资源管理器 NPM 的使用, webpack 的作用, vs code 开发工具的基本使用方法, ES6 常用的新特性语法

2. API 文档与模拟数据接口

章节内容：RESTful 架构, 运用 Swagger 编写 API 文档, Mock.js 基本语法, easyMock 实现模拟接口的编写

3. 使用 ElementUI 开发管理后台

章节内容：elementUI 提供的脚手架搭建管理后台的方法, elementUI 的 table 组件的使用和实现列表功能, elementUI 的 pagination 组件的使用和实现

分页功能, elementUI 的 form 相关组件的使用和实现条件查询功能, elementUI 的 dialog 组件和 \$message 的使用和实现弹出窗口和消息提示功能, elementUI 的 select 组件的使用和实现下拉列表功能, 新增数据和修改数据的功能, confirm 的使用和实现询问与现删除功能

路由与状态管理.

章节内容：路由在单页面工程中的作用, 可搜索下拉框.复合型输入框等 ElementUI 的使用 ,完成招聘管理功能, 文章管理功能, Vuex 状态管理在工程中的作用

4. 网站前台

章节内容：Nuxt 框架的基本使用方法, 十次方网站前台的搭建, 十次方网站前台活动模块的功能, 十次方网站前台招聘模块的功能, 用户注册功能, js-cookie 的使用, 微信扫码登陆的功能, nuxt 嵌套路由的使用, 吐槽列表与详细页, 发吐槽与评论功能, 问答频道功能, DataURL 和阿里云 OSS

5. 系统设计与工程搭建

章节内容：十次方的需求分析, 十次方的系统设计, 项目的前期准备工作(配置 JDK 与 本地仓库), 十次方父模块与公共模块的搭建, 基础微服务-标签 CRUD 的功能, 掌握公共异常处理类

6. 微服务功能开发

章节内容：基础微服务的开发, 招聘微服务的开发, 问答微服务的开发, 文章微服务的开发, SpringCache 与 SpringDataRedis 的使用

7. 文档型数据库 MongoDB

章节内容：MongoDB 的特点和体系结构, 常用的 MongoDB 命令,使用运用 Java 操作 MongoDB, 使用 SpringDataMongoDB 完成吐槽微服务的开发, 使用 SpringDataMongoDB 完成评论系统的开发

8. 消息中间件 RabbitMQ 与搜索微服务

章节内容 消息队列的应用场景以及 RabbitMQ 的主要概念, RabbitMQ 安装以及 RabbitMQ 三种模式的入门案例, 用户注册, 能够将消息发送给 RabbitMQ, 短信微服务, 能够接收消息并调用阿里云通信完成短信发送, 搜索微服务

9. 密码加密与微服务鉴权 JWT

章节内容：BCrypt 密码加密算法实现注册与登陆功能, 常见的认证机制, JWT 的组成部分以及使用 JWT 的优点, 使用 JWT 创建和解析 token, 使用 JWT 完成微服务鉴权

10. SpringCloud 初入江湖&SpringCloud 一统天下

章节内容：SpringCloud 包含的主要框架, SpringCloud 包含的主要框架, 使用服务发现组件 Eureka, 使用 Feign 实现服务间的调用, 交友微服务开发, 在项目中使用 Hystrix 实现微服务熔断, 在项目中使用 Zuul 实现微服务网关, 在项目中使用 SpringCloudConfig 实现配置集中管理, 在项目中使用 SpringCloudBus 实现配置的在线更新

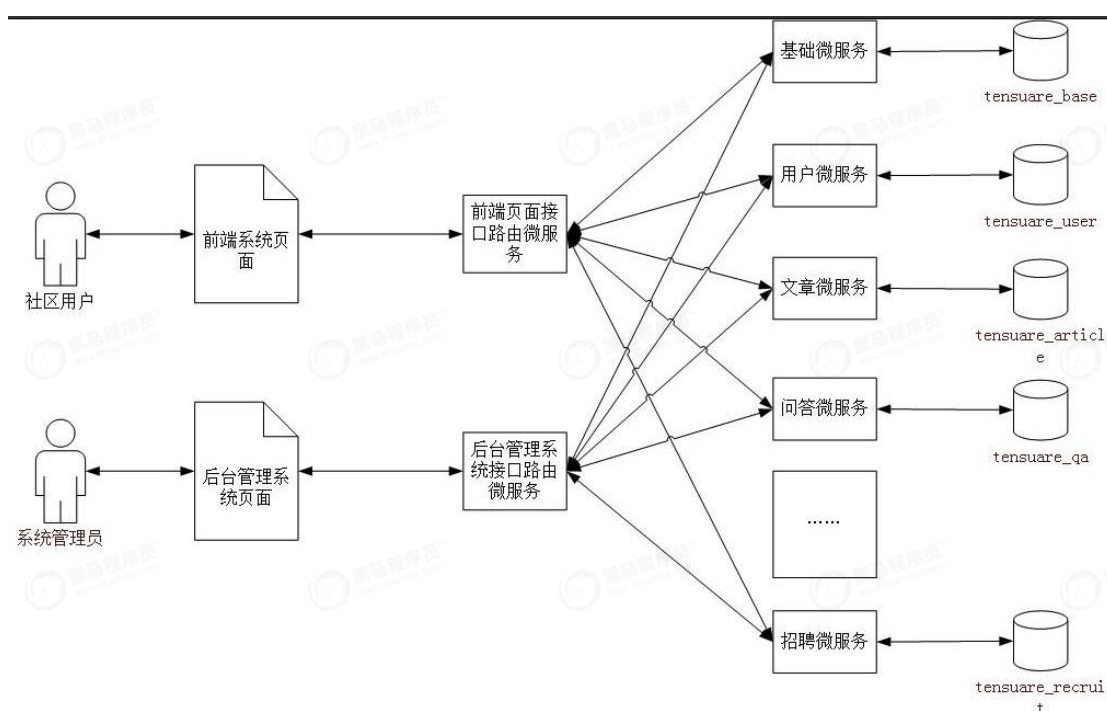
11. 爬虫

章节内容：什么是网络爬虫, 网络爬虫可以做什么, 网络爬虫常用的技术, 爬虫框架 Webmagic, 十次方文章数据爬取, 十次方用户数据爬取,

12. 人工智能

章节内容：学习目标, 人工智能与机器学习, 智能分类, IK 分词器, Deeplearning4j 之 Word2VEC, 构建卷积神经网络模型, 实现智能分类

2. 十次方项目技术架构



3. 十次方项目面试常见问题

3.1 对微服务有何了解？

微服务，又称微服务架构，是一种架构风格，它将应用程序构建为以业务领域为模型的小型自治服务集合。

通俗的来说就是将业务模块区分，不受编程语言限制，提供更好的可扩展性，独立开发部署。一个业务模块的改动不会影响到其他业务。可以单独处理每个服务组件的问题，而对整个应用程序没有影响或影响最小。

3.2 微服务架构有哪些优势？

独立开发 – 所有微服务都可以根据各自的功能轻松开发

独立部署 – 基于其服务，可以在任何应用程序中单独部署它们

故障隔离 – 即使应用程序的一项服务不起作用，系统仍可继续运行

混合技术堆栈 – 可以使用不同的语言和技术来构建同一应用程序的不同服务

粒度缩放 – 单个组件可根据需要进行缩放，无需将所有组件缩放在一起

3.3 微服务有哪些特点？

解耦 – 系统内的服务很大程度上是分离的。因此，整个应用程序可以轻松构建，更改和扩展

组件化 – 微服务被视为可以轻松更换和升级的独立组件

业务能力 – 微服务非常简单，专注于单一功能

自治 – 开发人员和团队可以彼此独立工作，从而提高速度

持续交付 – 通过软件创建，测试和批准的系统自动化，允许频繁发布软件

责任 – 微服务不关注应用程序作为项目。相反，他们将应用程序视为他们负责的产品

分散治理 – 重点是使用正确的工具来做正确的工作。这意味着没有标准化模式或任何技术模式。开发人员可以自由选择最有用的工具来解决他们的问题

敏捷 – 微服务支持敏捷开发。任何新功能都可以快速开发并再次丢弃

3.4 微服务架构如何运作？

客户端 – 来自不同设备的不同用户发送请求。

身份提供商 – 验证用户或客户身份并颁发安全令牌。

API 网关 – 处理客户端请求。

静态内容 – 容纳系统的所有内容。

管理 – 在节点上平衡服务并识别故障。

服务发现 – 查找微服务之间通信路径的指南。

内容交付网络 – 代理服务器及其数据中心的分布式网络。

远程服务 – 启用驻留在 IT 设备网络上的远程访问信息。

3.5 微服务架构的优缺点是什么？

微服务架构的优点	微服务架构的缺点
自由使用不同的技术	增加故障排除挑战
每个微服务都侧重于单一功能	由于远程呼叫而增加延迟
支持单个可部署单元	增加了配置和其他操作的工作量
允许经常发布软件	难以保持交易安全
确保每项服务的安全性	艰难地跨越各种边界跟踪数据
多个服务是并行开发和部署的	难以在服务之间进行编码

3.6 单片，SOA 和微服务架构有什么区别？

单片架构类似于大容器，其中应用程序的所有软件组件组装在一起并紧密封装。

一个**面向服务**的架构是一种相互通信服务的集合。通信可以涉及简单的数据传递，也可以涉及两个或多个协调某些活动的服务。

微服务架构是一种架构风格,它将应用程序构建为以业务域为模型的小型自治服务集合。

3.7 在使用微服务架构时,您面临哪些挑战?

开发一些较小的微服务听起来很容易,但开发它们时经常遇到的挑战如下。

自动化组件:难以自动化,因为有许多较小的组件。因此,对于每个组件,我们必须遵循 Build, Deploy 和 Monitor 的各个阶段。

易感性:将大量组件维护在一起变得难以部署,维护,监控和识别问题。它需要在所有组件周围具有很好的感知能力。

配置管理:有时在各种环境中维护组件的配置变得困难。

调试:很难找到错误的每一项服务。维护集中式日志记录和仪表板以调试问题至关重要。

3.8 SOA 和微服务架构之间的主要区别是什么?

SOA	微服务
遵循“尽可能多的共享”架构方法	遵循“尽可能少分享”的架构方法
重要性在于 业务功能 重用	重要性在于“有界背景”的概念
他们有 共同的 治理 和标准	他们专注于 人们的 合作 和其他选择的自由
使用 企业服务总线 (ESB) 进行通信	简单的消息系统
它们支持 多种消息协议	他们使用 轻量级协议,如 HTTP / REST 等。
多线程,有更多的开销来处理 I / O.	单线程 通常使用 Event Loop 功能进行非锁定 I / O 处理

最大化应用程序服务可重用性	专注于 解耦
传统的关系数据库 更常用	现代 关系数据库 更常用
系统的变化需要修改整体	系统的变化是创造一种新的服务
DevOps / Continuous Delivery 正在变得流行，但还不是主流	专注于 DevOps /持续交付

3.9 什么是耦合？

组件之间依赖关系强度的度量被认为是**耦合**。一个好的设计总是被认为具有**高内聚力**和**低耦合性**。

3.10 什么是 Rest/ Restful 架构以及它的用途是什么？

Rest/ Restful Web 服务是一种帮助计算机系统通过 Internet 进行通信的架构风格。这使得微服务更容易理解和实现。

微服务可以使用或不使用 RESTful API 实现，但使用 RESTful API 构建松散耦合的微服务总是更容易。

Restful 架构，就是目前最流行的一种互联网软件架构。它结构清晰.符合标准.易于理解.扩展方便，所以正得到越来越多网站的采用。REST 这个词，是 Roy Thomas Fielding 在他 2000 年的博士论文中提出的。

Rest 是 Representational State Transfer 的缩写，翻译是“表现层状态转化”。

可以总结为一句话:REST 是所有 Web 应用都应该遵守的架构设计指导原则。

面向资源是 Rest 最明显的特征，对于同一个资源的一组不同的操作。资源是服务器 上一个可命名的抽象概念，资源是以名词为核心来组织的，首先关注的是名词。REST 要求，必须通过统一的接口来对资源执行各种操作。对于每个资源只能执行一组有限的操 作。

7 个 HTTP 方法：GET/POST/PUT/DELETE/PATCH/HEAD/OPTIONS。

3.11 你对 SpringBoot 有什么了解？

事实上，随着新功能的增加，弹簧变得越来越复杂。如果必须启动新的 spring 项目，则必须添加构建路径或添加 maven 依赖项，配置应用程序服务器，添加 spring 配置。所以一切都必须从头开始。

SpringBoot 是解决这个问题的方法。使用 SpringBoot 可以避免所有样板代码和 XML 配置。快速构建项目。感觉是 SpringMVC 精简版。

3.12 什么是 Spring 引导的执行器？

SpringBoot 执行程序提供了 restful Web 服务，以访问生产环境中运行应用程序的当前状态。在执行器的帮助下，您可以检查各种指标并监控您的应用程序。

3.13 什么是 SpringCloud？

根据 SpringCloud 的官方网站，SpringCloud 为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智能路由，领导选举，分布式会话，集群状态）。

3.14 SpringCloud 解决了哪些问题？

在使用 SpringBoot 开发分布式微服务时，我们面临的问题很少由 SpringCloud 解决。

与分布式系统相关的复杂性 – 包括网络问题，延迟开销，带宽问题，安全问题。

处理服务发现的能力 – 服务发现允许集群中的进程和服务找到彼此并进行通信。

解决冗余问题 – 冗余问题经常发生在分布式系统中。

负载均衡 – 改进跨多个计算资源（例如计算机集群，网络链接，中央处理单元）的工作负载分布。

减少性能问题 – 减少因各种操作开销导致的性能问题。

3.15 在 Spring MVC 应用程序中使用 WebMvcTest 注释有什么用处？

```
@WebMvcTest(value = ToTestController.class, secure = false):
```

在测试目标只关注 Spring MVC 组件的情况下，**WebMvcTest** 注释用于单元测试 Spring MVC 应用程序。在上面显示的快照中，我们只想启动 ToTestController。执行此单元测试时，不会启动所有其他控制器和映射。

3.16 你能否给出关于休息和微服务的要点？

休息

虽然您可以通过多种方式实现微服务，但 REST over HTTP 是实现微服务的一种方式。REST 还可用于其他应用程序，如 Web 应用程序，API 设计和 MVC 应用程序，以提供业务数据。

微服务

微服务是一种体系结构，其中系统的所有组件都被放入单独的组件中，这些组件可以单独构建，部署和扩展。微服务的某些原则和最佳实践有助于构建弹性应用程序。

简而言之，您可以说 REST 是构建微服务的媒介。

3.17 什么是不同类型的微服务测试？

在使用微服务时，由于有多个微服务协同工作，测试变得非常复杂。因此，测试分为不同的级别。

在**底层**，我们有面向技术的测试，如单元测试和性能测试。这些是完全自动化的。

在**中间层面**，我们进行了诸如压力测试和可用性测试之类的探索性测试。

在**顶层**，我们的验收测试数量很少。这些验收测试有助于利益相关者理解和验证软件功能。

3.18 您对 Distributed Transaction 有何了解？

分布式事务是指单个事件导致两个或多个不能以原子方式提交的单独数据源的突变的任何情况。在微服务的世界中，它变得更加复杂，因为每个服务都是一个工作单元，并且大多数时候多个服务必须协同工作才能使业务成功。

3.19 什么是 Idempotence 以及它在哪里使用？

幂等性是能够以这样的方式做两次事情的特性，即最终结果将保持不变，即好像它只做了一次。

用法：在远程服务或数据源中使用 Idempotence，这样当它多次接收指令时，它只处理指令一次。

3.20 什么是有界上下文？

有界上下文是域驱动设计的核心模式。DDD 战略设计部门的重点是处理大型模型和团队。DDD 通过将大型模型划分为不同的有界上下文并明确其相互关系来处理大型模型。

3.21 什么是双因素身份验证？

双因素身份验证为帐户登录过程启用第二级身份验证。

因此，假设用户必须只输入用户名和密码，那么这被认为是单因素身份验证。

3.22 双因素身份验证的凭据类型有哪些？

- 1 Something you know - ex: PIN, password or a pattern
- 2 Something you have - ex: ATM card, phone or OTP
- 3 Something you are – ex: Biometric fingerprint or voice print

3.23 什么是客户证书？

客户端系统用于向远程服务器发出经过身份验证的请求的一种数字证书称为**客户端证书**。客户端证书在许多相互认证设计中起着非常重要的作用，为请求者的身份提供了强有力的保证。比如 JWT 鉴权等。

4. 十次方技术点

1. MongoDB

1.1 什么是 MongoDB

MongoDB 是一个基于分布式文件存储的数据库。由 C++ 语言编写。旨在为 WEB 应用提供可扩展的高性能数据存储解决方案。MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。它支持的数据结构非常松散，是类似 json 的 bson 格式，因此可以存储比较复杂的数据类型。

1.2 MongoDB 特点

Mongo 最大的特点是它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

它的特点是高性能、易部署、易使用，存储数据非常方便。主要功能特性有：

1. 面向集合存储，易存储对象类型的数据。
2. 模式自由。
3. 支持动态查询。
4. 支持完全索引，包含内部对象。
5. 支持复制和故障恢复。
6. 使用高效的二进制数据存储，包括大型对象（如视频等）。
7. 自动处理碎片，以支持云计算层次的扩展性。
8. 支持 RUBY，PYTHON，JAVA，C++，PHP，C#等多种语言。

9. 文件存储格式为 BSON (一种 JSON 的扩展)。

1.3 MongoDB 体系结构

MongoDB 的逻辑结构是一种层次结构。主要由：文档(document).集合(collection).数据库(database)这三部分组成的。逻辑结构是面向用户的，用户使用 MongoDB 开发应用程序使用的就是逻辑结构。

1. MongoDB 的文档 (document)，相当于关系数据库中的一行记录。
2. 多个文档组成一个集合 (collection)，相当于关系数据库的表。
3. 多个集合 (collection)，逻辑上组织在一起，就是数据库 (database)。
4. 一个 MongoDB 实例支持多个数据库 (database)。

1.4 MongoDB 数据类型

数据类型	描述
String	字符串。存储数据常用的数据类型。在 MongoDB 中，UTF-8 编码的字符串才是合法的。
Integer	整型数值。用于存储数值。根据你所采用的服务器，可分为 32 位或 64 位。
Boolean	布尔值。用于存储布尔值（真/假）。
Double	双精度浮点值。用于存储浮点值。
Array	用于将数组或列表或多个值存储为一个键。
Timestamp	时间戳。记录文档修改或添加的具体时间。
Object	用于内嵌文档。
Null	用于创建空值。
Date	日期时间。用 UNIX 时间格式来存储当前日期或时间。你可以指定自己的日期时间：创建 Date 对象，传入年月日信息。
Object ID	对象 ID。用于创建文档的 ID。
Binary Data	二进制数据。用于存储二进制数据。
Code	代码类型。用于在文档中存储 JavaScript 代码。
Regular expression	正则表达式类型。用于存储正则表达式。

2.即时通讯

2.1 什么是即时通信？

即时通信 (Instant Messaging，简称 IM) 是一个允许两人或多人使用网络实时的传递文字消息.文件.语音与视频交流。即时通讯技术应用于需要**实时**收发消息的业务场景。即时通讯使用的是长连接。

2.2 什么是短连接？

客户端和服务端每进行一次通讯，就建立一次连接，通讯结束就中断连接。

HTTP 是一个简单的请求-响应协议，它通常运行在 TCP 之上。HTTP/1.0 使用的 TCP 默认是短连接。

2.3 什么是长连接？

是指在建立连接后可以连续多次发送数据，直到双方断开连接。

2.4 短连接和长连接的区别？

2.4.1 通讯流程

短连接：创建连接 -> 传输数据 -> 关闭连接

长连接：创建连接 -> 传输数据 -> 保持连接 -> 传输数据 -> ->

关闭连接

2.4.2 适用场景

短连接：并发量大，数据交互不频繁情况

长连接：数据交互频繁，点对点的通讯

2.4.3 通讯方式

方式	说明
短连接	我跟你发信息，必须等到你回复我或者等了一会等不下去了，就结束通讯了
长连接	我跟你发信息，一直保持通讯，在保持通讯这个时段，我去做其他事情的当中你回复我了， 续做事

3. websocket 协议

3.1 何为 websocket 协议？

WebSocket 是 HTML5 开始提供的一种在单个 TCP 连接上进行全双工通讯的协议。

何谓全双工：全双工（Full Duplex）是通讯传输的一个术语。双方在通信时允许数据在两个方向上同时传输，它在能力上相当于两个单工通信方式的结合。全双工指可以同时进行信号的双向传输。指 A→B 的同时 B→A，就像是双向车道。单工就就像是汽车的单行道，是在只允许甲方向乙方传送信息，而乙方不能向甲方传送。

在 WebSocket 中，浏览器和服务器只需要完成一次握手，就可以创建持久性的连接，并进行双向数据传输。

在推送功能的实现技术上，相比使用 Ajax 定时轮询的方式(setInterval)，WebSocket 更节省服务器资源和带宽。

服务器向客户端发送数据的功能是 websocket 协议的典型使用场景

3.2 websocket 常用事件方法？

WebSocket 事件

以下是 WebSocket 对象的相关事件。假定我们使用了以上代码创建了 Socket 对象：

事件	事件处理程序	描述
open	Socket.onopen	连接建立时触发
message	<u>Socket.onmessage</u>	客户端接收服务端数据时触发
error	Socket.onerror	通信发生错误时触发
close	<u>Socket.onclose</u>	连接关闭时触发

WebSocket 方法

方法	描述
Socket.send()	使用连接发送数据
Socket.close()	关闭连接

4.接口加密

4.1 加密方式

4.1.1 摘要算法

消息摘要就是把任意长度的输入揉和而产生长度固定的信息。

消息摘要算法的主要特征是加密过程不需要密钥，并且经过加密的数据无法被解密，只有输入相同的明文数据经过相同的消息摘要算法才能得到相同的密文。消息摘要算法不存在密钥的管理与分发问题，适合于分布式网络上使用。

消息摘要的主要特点有：

无论输入的消息有多长，计算出来的消息摘要的长度总是固定的。

消息摘要看起来是“随机的”。这些数据看上去是胡乱的杂凑在一起的。

只要输入的消息不同，对其进行摘要后产生的摘要消息也必不相同；但相同的输入必会产生相同的输出。

只能进行正向的消息摘要，而无法从摘要中恢复出任何消息，甚至根本就找不到任何与原信息相关的信息。

虽然“碰撞”是肯定存在的，但好的摘要算法很难能从中找到“碰撞”。即无法找到两条不同消息，但是它们的摘要相同。

常见的摘要算法：CRC.MD5.SHA 等

4.1.2 对称加密

加密和解密使用相同密钥的加密算法。

对称加密的特点：

速度快，通常在消息发送方需要加密大量数据时使用。

密钥是控制加密及解密过程的指令。

算法是一组规则，规定如何进行加密和解密。

典型应用场景：离线的大量数据加密（用于存储的）

常用的加密算法：

DES.3DES.AES.TDEA.Blowfish.RC2.RC4.RC5.IDEA.SKIPJACK 等。

加密的安全性不仅取决于加密算法本身，密钥管理的安全性更是重要。如何把密钥安全地传递到解密者手上就成了必须要解决的问题。

4.1.3 非对称加密

非对称加密算法是一种**密钥**的保密方法，加密和解密使用两个不同的密钥，公开密钥（publickey:简称公钥）和私有密钥（privatekey:简称私钥）。公钥与私钥是一对，如果用公钥对数据进行加密，只有用对应的私钥才能解密。

非对称加密算法的特点：

- 算法强度复杂
- 加密解密速度没有对称密钥算法的速度快

经典应用场景：数字签名（公钥加密，私钥验证）

常用的算法 **RSA.Elgamal.背包算法.Rabin.D-H.ECC**(椭圆曲线加密算法)。

4.1.4 数字签名

数字签名（又称公钥数字签名）是一种类似写在纸上的普通的物理签名，是使用了公钥加密领域的技术实现，用于鉴别数字信息的方法。

数字签名通常使用私钥生成签名，使用公钥验证签名。

签名及验证过程：

1. 发送方用一个哈希函数（例如 MD5）从报文文本中生成报文摘要，然后用自己的私钥对这个摘要进行加密

2. 将加密后的摘要作为报文的数字签名和报文一起发送给接收方
3. 接收方用与发送方一样的哈希函数从接收到的原始报文中计算出报文摘要，
4. 接收方再用发送方的公用密钥来对报文附加的数字签名进行解密
5. 如果这两个摘要相同.接收方就能确认该数字签名是发送方的。

数字签名验证的两个作用：

- 确定消息确实是由发送方签名并发出来的
- 确定消息的完整性

5. 消息通知

5.1 消息通知的业务场景

消息通知微服务的定位是“平台内”的“消息”功能，分为全员消息，订阅类消息，点对点消息。例如系统通知，私信，@类消息

全员消息

系统通知，活动通知，管理员公告等全部用户都会收到的消息

订阅类消息

关注某一类数据的用户，该类数据有更新时向用户发送的消息。例如关注某位大v的微博，公众号，订阅某位知名作家的专栏

点对点消息

某位用户对另外一位用户进行操作后，系统向被操作的用户发送的消息。例如点赞，发红包。

5.2 消息通知与即时通讯的区别

即时通信

传输的内容包括文字聊天、语音消息发送、文件传输、音视频播放，内容极其丰富。

核心需求点 要求连接稳定可靠。就像网络游戏，如果总是掉线，你还玩的下去吗？

系统建设成本 存储成本高（图片、视频等）。基于TCP协议，需建设或租用多线机房，甚至自建成本。

交互方式 任何消息均可回复

技术实现 XMPP, MQTT, Strophe等全双工长连接协议

消息通知

以文字、超链接为主，辅以图片，不能再多了。

要求消息的高送达率，也就是说“这事儿一定要想尽办法通知到对方”。对延时要求不高。

一般只保存文本消息，存储成本低。可根据用户量自由调整服务器集群配置。

消息一般被设计为“仅通知，不需要回复”

JMS, AMQP, http等各种协议

5.3 消息通知存在的问题

5.3.1 数据库访问压力大

用户的通知消息和新通知提醒数据都放在数据库中，数据库的读写操作频繁，尤其是 tb_notice_refresh 表，访问压力大。

5.3.2 服务器性能压力大

采用页面轮询调接口的方式实现服务器向用户推送消息通知，服务器的接口访问压力大。

5.3.3 改进的方法

使用 rabbitmq 实现新消息提醒数据的缓存功能，替代 tb_notice_refresh 表

使用全双工长连接的方式实现服务器向用户推送最新的消息通知，替换轮询

- 页面使用 websocket
- 微服务端使用异步高性能框架 netty

6.Redis 分布式缓存

6.1 Redis 读写分离

单机 Redis 的读写速度非常快，能够支持大量用户的访问。虽然 Redis 的性能很高，但是对于大型网站来说，每秒需要获取的数据远远超过单台 Redis

服务所能承受的压力,所以我们迫切需要一种方案能够解决单台 Redis 服务性能不足的问题。

6.2 Redis 性能测试

6.2.1 Redis-benchmark

Redis-benchmark 是官方自带的 Redis 性能测试工具,用来测试 Redis 在当前环境下的读写性能。我们在使用 Redis 的时候,服务器的硬件配置、网络状况、测试环境都会对 Redis 的性能有所影响,我们需要对 Redis 实时测试以确定 Redis 的实际性能。

6.2.2 TPS.QPS.RT

响应时间(RT)

响应时间是指系统对请求作出响应的时间。

直观上看,这个指标与人对软件性能的主观感受是非常一致的,因为它完整地记录了整个计算机系统处理请求的时间。由于一个系统通常会提供许多功能,而不同功能的业务逻辑也千差万别,因而不同功能的响应时间也不尽相同。

在讨论一个系统的响应时间时,通常是指该系统所有功能的平均时间或者所有功能的最大响应时间。吞吐量 TPS 吞吐量是指系统在单位时间内处理请求的数量。

对于一个多用户的系统,如果只有一个用户使用系统的平均响应时间是 t ,当有 n 个用户使用,每个用户看到的响应时间通常并不是 $n \times t$,而往往比 $n \times t$ 小很多。这是因为在处理单个请求时,在每个时间点都可能有许多资源被闲置,当处理多个请求时,如果资源配置合理,每个用户看到的平均响应时间并不随用户数的增加而线性增加。

实际上，不同系统的平均响应时间随用户数增加而增长的速度也不大相同，这也是采用吞吐量来度量并发系统的性能的主要原因。一般而言，吞吐量是一个比较通用的指标，两个具有不同用户数和用户使用模式的系统，如果其最大吞吐量基本一致，则可以判断两个系统的处理能力基本一致。

举例来说系统能处理的

并发数 100

响应时间(RT) 100ms

每秒的 TPS

一个并发，1 秒就是 $1000/100 = 10$ TPS

十个并发，1 秒就是 $1000/100 * 10 = 100$ TPS

百个并发，1 秒就是 $1000/100 * 100 = 1000$ TPS

以上是资源配置合理的情况

如果系统能处理 100 并发，却有 200 个并发请求

1 秒就是 $1000/(100 \text{ 之前的响应时间} + \text{系统处理多出的并发消耗掉的时间} 100) * 100 = 500$ TPS

总结：吞吐量必然和当前系统是否配置合理有关，超过最大承受后，吞吐量会下降，生活中想想颐和园的售票处

每秒查询率 QPS

每秒查询率 QPS 是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准，在互联网中，经常用每秒查询率来衡量服务器的性能。对应 fetches/sec，即每秒的响应请求数，也即是最大吞吐能力。

TPS 和 QPS 区别：

TPS 牵扯到系统的整体性能，如果只是想了解 Redis 的性能就会询问 QPS 不牵扯，系统架构，业务性能等因素

6.3 Redis 同步原理

Redis 的主从复制，主服务器执行写操作命令，从服务器会通过主服务器的数据的变化，同步数据到从服务器。但是如果主服务器下线，从服务器无法连接主服务器，那么数据同步该如何拿到不能连接主服务器这段时间的命令呢？

主从复制中的主从服务器双方的数据库将保存相同的数据，概念上将这种现象称作数据库状态一致。

Redis 数据库持久化有两种方式：RDB 全量持久化和 AOF 增量持久化。

通过上面的例子，我们知道 Redis 的主从复制，主服务器执行写操作命令，从服务器会通过主服务器的数据的变化，同步数据到从服务器。但是如果主服务器下线，从服务器无法连接主服务器，那么数据同步该如何拿到不能连接主服务器这段时间的命令呢？

主从复制中的主从服务器双方的数据库将保存相同的数据，概念上将这种现象称作**数据库状态一致**。

Redis 数据库持久化有两种方式：RDB 全量持久化和 AOF 增量持久化。

6.4 Redis 的高可用 Sentinel

高可用是分布式系统架构设计中必须考虑的因素之一，它是通过架构设计减少系统不能提供服务的时间。保证高可用通常遵循下面几点：

1. 单点是系统高可用的大敌，应该尽量在系统设计的过程中避免单点。
2. 通过架构设计而保证系统高可用的，其核心准则是：冗余。

3. 每次出现故障需要人工介入恢复，会增加系统不可用的时间，实现自动故障转移。

7.JUC 多线程

7.1 认识多线程

一个采用了多线程技术的应用程序可以更好地利用系统资源。其主要优势在于充分利用了 CPU 的空闲时间片，可以用尽可能少的时间来对用户的要求做出响应，使得进程的整体运行效率得到较大提高，同时增强了应用程序的灵活性。

更为重要的是，由于同一进程的所有线程是共享同一内存，所以不需要特殊的数据传送机制，不需要建立共享存储区或共享文件，从而使得不同任务之间的协调操作与运行数据的交互资源的分配等问题更加易于解决。

7.2 什么是线程？

进程内部的一个独立执行单元；一个进程可以同时并发的运行多个线程，可以理解为一个进程便相当于一个单 CPU 操作系统，而线程便是这个系统中运行的多个任务。

7.3 什么是进程？

是指一个内存中运行的应用程序，每个进程都有一个独立的内存空间，一个应用程序可以同时运行多个进程；进程也是程序的一次执行过程，是系统运行程序的基本单位；系统运行一个程序即是一个进程从创建运行到消亡的过程。

7.4 进程与线程的区别？

进程：有独立的内存空间，进程中的数据存放空间（堆空间和栈空间）是独立的，至少有一个线程。

线程：堆空间是共享的，栈空间是独立的，线程消耗的资源比进程小的多。

注意：

1. 因为一个进程中的多个线程是并发运行的，那么从微观角度看也是有先后顺序的，哪个线程执行完全取决于 CPU 的调度，程序员是不能完全控制的（可以设置线程优先级，有限度的控制顺序）。而这也就造成的多线程的随机性。

2. Java 程序的进程里面至少包含两个线程，主线程也就是 main() 方法线程，另外一个垃圾回收机制线程。每当使用 java 命令执行一个类时，实际上都会启动一个 JVM，每一个 JVM 实际上就是在操作系统中启动了一个线程，java 本身具备了垃圾的收集机制，所以在 Java 运行时至少会启动两个线程。

3. 由于创建一个线程的开销比创建一个进程的开销小的多，那么我们在开发多任务运行的时候，通常考虑创建多线程，而不是创建多进程。

7.4 什么是线程安全？

线程安全问题都是由全局变量及静态变量引起的。若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。

7.5 什么时候使用线程同步？

当我们使用多个线程访问同一资源的时候，且多个线程中对资源有写的操作，就容易出现线程安全问题。要解决上述多线程并发访问一个资源的安全问题，Java 中提供了同步机制(synchronized)来解决。

7.6 线程同步机制有哪些？

同步代码块.同步方法.lock 锁等。

7.7 对死锁的理解？

多线程死锁：同步中嵌套同步,导致锁无法释放。

死锁解决办法：不要在同步中嵌套同步

7.8 多线程并发的 3 个特性？

原子性，可见性和有序性。

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行

可见性：当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值

有序性：程序执行的顺序按照代码的先后顺序执行

7.9 锁优化

synchronized 是重量级锁，效率不高。但在 JDK 1.6 中对 synchronize 的实现进行了各种优化，使得它显得不是那么重了。JDK1.6 对锁的实现引入了大量的优化，如自旋锁.适应性自旋锁.锁消除.锁粗化.偏向锁.轻量级锁等技术来减少锁操作的开销。

锁主要存在四种状态，依次是：无锁状态.偏向锁状态.轻量级锁状态.重量级锁状态，他们会随着竞争的激烈而逐渐升级。

注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

7.9.1 自旋锁

线程的阻塞和唤醒需要 CPU 从用户态转为核心态,频繁的阻塞和唤醒对 CPU 来说是一件负担很重的工作,势必会给系统的并发性能带来很大的压力。同时我们发现在许多应用上面,对象锁的锁状态只会持续很短一段时间,为了这一段很短的时间频繁地阻塞和唤醒线程是非常不值得的。所以引入自旋锁。

所谓自旋锁,就是让该线程等待一段时间,不会被立即挂起,看持有锁的线程是否会很快释放锁。怎么等待呢?执行一段无意义的循环即可(自旋)。

自旋等待不能替代阻塞,虽然它可以避免线程切换带来的开销,但是它占用了处理器的时间。如果持有锁的线程很快就释放了锁,那么自旋的效率就非常好,反之,自旋的线程就会白白消耗掉处理的资源,它不会做任何有意义的工作,典型的占着茅坑不拉屎,这样反而会带来性能上的浪费。所以说,自旋等待的时间(自旋的次数)必须要有一个限度,如果自旋超过了定义的时间仍然没有获取到锁,则应该被挂起。

自旋锁在 JDK 1.4.2 中引入,默认关闭,但是可以使用 `-XX:+UseSpinning` 开启,在 JDK1.6 中默认开启。同时自旋的默认次数为 10 次,可以通过参数 `-XX:PreBlockSpin` 来调整;

如果通过参数 `-XX:preBlockSpin` 来调整自旋锁的自旋次数,会带来诸多不便。假如我将参数调整为 10,但是系统很多线程都是等你刚刚退出的时候就释放了锁(假如你多自旋一两次就可以获取锁),你是不是很尴尬。于是 JDK1.6 引入自适应的自旋锁,让虚拟机会变得越来越聪明。

7.9.2 适应自旋锁

JDK 1.6 引入了更加聪明的自旋锁，即自适应自旋锁。所谓自适应就意味着自旋的次数不再是固定的，它是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。它怎么做呢？线程如果自旋成功了，那么下次自旋的次数会更加多，因为虚拟机认为既然上次成功了，那么此次自旋也很有可能会再次成功，那么它就会允许自旋等待持续的次数更多。反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要或者这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源。

有了自适应自旋锁，随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测会越来越准确，虚拟机会变得越来越聪明。

7.9.3 锁消除

为了保证数据的完整性，我们在进行操作时需要对这部分操作进行同步控制，但是在有些情况下，JVM 检测到不可能存在共享数据竞争，这是 JVM 会对这些同步锁进行锁消除。锁消除的依据是逃逸分析的数据支持。

如果不存在竞争，为什么还需要加锁呢？所以锁消除可以节省毫无意义的请求锁的时间。变量是否逃逸，对于虚拟机来说需要使用数据流分析来确定，但是对于我们程序员来说这还不清楚么？我们会在明明知道不存在数据竞争的代码块前加上同步吗？但是有时候程序并不是我们所想的那样？我们虽然没有显示使用锁，但是我们在一些 JDK 的内置 API 时，如 `StringBuffer`、`Vector`、`HashTable` 等，这个时候会存在隐形的加锁操作。比如 `StringBuffer` 的 `append()` 方法，`Vector` 的 `add()` 方法：

7.9.4 锁粗化

在使用同步锁的时候，需要让同步块的作用范围尽可能小，仅在共享数据的实际作用域中才进行同步，这样做的目的是为了使需要同步的操作量尽可能缩小，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁。

在大多数的情况下，上述观点是正确的。但是如果一系列的连续加锁解锁操作，可能会导致不必要的性能损耗，所以引入锁粗化的概念。

锁粗化概念比较好理解，就是将多个连续的加锁.解锁操作连接在一起，扩展成一个范围更大的锁。如上面实例：vector 每次 add 的时候都需要加锁操作，JVM 检测到对同一个对象（vector）连续加锁.解锁操作，会合并一个更大范围的加锁.解锁操作，即加锁解锁操作会移到 for 循环之外。

7.9.5 偏向锁

轻量级锁的加锁解锁操作是需要依赖多次 CAS 原子指令的。而偏向锁只需要检查是否为偏向锁.锁标识为以及 ThreadID 即可，可以减少不必要的 CAS 操作。

7.9.6 轻量级锁

引入轻量级锁的主要目的是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。当关闭偏向锁功能或者多个线程竞争偏向锁导致偏向锁升级为轻量级锁，则会尝试获取轻量级锁。轻量级锁主要使用 CAS 进行原子操作。

但是对于轻量级锁，其性能提升的依据是“对于绝大部分的锁，在整个生命周期内都是不会存在竞争的”，如果打破这个依据则除了互斥的开销外，还有额外的 CAS 操作，因此在有多线程竞争的情况下，轻量级锁比重量级锁更慢。

7.9.7 重量锁

重量级锁通过对象内部的监视器 (monitor) 实现 , 其中 monitor 的本质是依赖于底层操作系统的 Mutex Lock (互斥锁) 实现 , 操作系统实现线程之间的切换需要从用户态到内核态的切换 , 切换成本非常高。